MofNeuroSim: Bit-Exact IEEE Floating-Point Arithmetic via Pure Spiking Neural Networks

Zhengzheng Tang [1]

Abstract

We present MofNeuroSim, a framework that implements IEEE 754 floating-point arithmetic entirely through spiking neural networks (SNNs) using Generalized Leaky Integrate-and-Fire (GLIF) neurons. Unlike conventional neuromorphic approaches that trade precision for efficiency, our method achieves **bit-exact** results by encoding floating-point numbers as binary pulse sequences and implementing all arithmetic operations through SNN logic gates. For FP16, FP32, and FP64, we achieve 0 ULP error—our results are bit-identical to IEEE 754 machine precision; for FP8, bit-exactness is achieved within the format's intrinsic limits. We prove that GLIF networks with soft reset and dynamic thresholds construct differential topological embeddings, establishing that GLIF neurons are mathematically equivalent to complex Hodgkin-Huxley neurons in computational capability—not a simplification. Our framework supports addition, multiplication, division, square root, and activation functions (Sigmoid, Tanh, GELU, Softmax). We further demonstrate end-to-end trainability via Straight-Through Estimator (STE) with a custom PulseSGD optimizer, and validate robustness under physical hardware conditions through LIF mode simulation.

1. Introduction

The computing world faces a dual crisis: the end of Moore's Law and the energy efficiency wall of von Neumann architectures (Schuman et al., 2017). As transistor scaling approaches physical limits and data movement dominates power consumption, the explosive computational demands of artificial intelligence require a paradigm shift. Spiking Neural Networks (SNNs), with their event-driven sparse computation and in-memory processing potential, have emerged as one of the most promising post-Moore computing paradigms, offering orders of magnitude improvement in energy efficiency (Kudithipudi et al., 2025).

Despite their promise for brain-inspired computing, SNNs have long been perceived as **inherently approximate computational models**. This "precision barrier" relegates SNNs to pattern recognition and sensory processing tasks, while high-fidelity computations—scientific simulations, financial modeling, and modern deep learning algorithms requiring precise gradients—must be outsourced to traditional coprocessors (George et al., 2019b; Dubey et al., 2020). This hybrid approach fundamentally compromises the vision of pure neuromorphic computing.

This work challenges the prevailing assumption that SNNs are inherently approximate. We demonstrate that the perceived precision limitation is not fundamental to spiking computation, but rather a consequence of conventional rate-coding and soft thresholding approaches. By leveraging binary pulse encoding with dynamic thresholds and soft reset mechanisms, SNNs can achieve bit-exact IEEE 754 floating-point arithmetic while preserving their energy efficiency advantages.

We present **MofNeuroSim**, a framework that implements IEEE 754 floating-point arithmetic entirely through spiking neural networks using Generalized Leaky Integrate-and-Fire (GLIF) neurons. The framework addresses three critical requirements for practical neuromorphic computing: (1) bit-exact precision matching traditional CPUs, (2) gradient-based trainability for deep learning integration, and (3) robustness under physical hardware constraints. Our contributions are:

1. **Theoretical Foundation**: We prove the *GLIF Topological Embedding Theorem*, establishing that GLIF networks with soft reset and dynamic thresholds create deterministic, chaos-free computation spaces homeomorphic to high-dimensional tori. This theorem has three profound implications:
    - It fundamentally overturns the "SNN is inherently fuzzy" perception, theoretically establishing the

[1]Boston University, Boston, MA, USA. Correspondence to: Zhengzheng Tang <zztangbu@bu.edu>.

Proceedings of the 43^{rd} International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

possibility of exact computation.

- It proves GLIF neurons are *topologically equivalent* to complex Hodgkin-Huxley neurons in computational capability—choosing GLIF is not a simplification sacrificing biological realism.
- It applies uniformly to ideal IF ($\beta = 1$) and physical LIF ($\beta < 1$) neurons, unifying exact computation with hardware physics.

2. **Complete Arithmetic System**: We construct a hierarchical architecture from GLIF neurons to logic gates, arithmetic units, IEEE 754 operators (FP8/16/32/64), and neural network layers including Linear, LayerNorm, RMSNorm, Attention, and activation functions (Sigmoid, Tanh, GELU, Softmax).

3. **Training Capability**: We implement end-to-end training via Straight-Through Estimator (STE) with a custom PulseSGD optimizer operating entirely in the pulse domain. The framework provides a TEMPORAL training mode interface for future integration of neuromorphic learning rules such as STDP.

4. **Hardware Robustness**: Through our neuron template system, we demonstrate that switching to LIF mode ($\beta < 1$) simulates physical imperfections (device noise, threshold variations, leakage currents), with theoretical guarantees from the embedding theorem ensuring computational stability.

5. **Bit-Exact Verification**: For FP16, FP32, and FP64, we achieve 0 ULP (Unit in the Last Place) error compared to PyTorch, meaning our results are bit-identical to IEEE 754 machine precision. For FP8, bit-exactness is achieved within the format's intrinsic limits.

The name "MofNeuroSim" reflects our vision of deep integration with Metal-Organic Framework (MOF) materials—emerging substrates that provide the *physical foundation* for high-density, low-power neuromorphic chips via MOF memristors (Xu et al., 2024; Bachinin et al., 2024). MofNeuroSim provides the "algorithmic soul" for these next-generation hardware platforms: a framework that is precise, trainable, and physically robust. Together, they form a complete theoretical loop from software algorithms to material hardware.

The remainder of this paper is organized as follows. Section 2 reviews related work in neuromorphic computing and SNN arithmetic. Section 3 presents our theoretical foundations and the complete MofNeuroSim architecture. Section 4 provides experimental validation. Section 5 discusses implications and future directions.

2. Related Work

2.1. Neuromorphic Computing and SNNs

Neuromorphic computing has emerged as a promising paradigm for energy-efficient computation (Schuman et al., 2017; Shrestha & Orchard, 2022). Spiking Neural Networks (SNNs) process information through discrete spike events rather than continuous activations, enabling event-driven computation with potential orders of magnitude improvement in energy efficiency (Kudithipudi et al., 2025). Comprehensive reviews of SNN architectures and applications are provided by Yamazaki et al. (2022), Rathi & Roy (2023), and Malcolm et al. (2023). Hardware implementations such as Intel's Loihi demonstrate the practical viability of neuromorphic processors (Davies et al., 2021).

Despite these advances, SNNs have been predominantly applied to pattern recognition and sensory processing tasks, with limited exploration of general-purpose computation. The perception of SNNs as inherently approximate computational models has restricted their application in domains requiring high-precision arithmetic.

2.2. Floating-Point Arithmetic in Neuromorphic Systems

Several works have explored implementing IEEE 754 floating-point operations on neuromorphic hardware. George et al. (2019a) proposed neuromorphic implementations of floating-point addition, while Dubey et al. (2020) addressed floating-point multiplication. Wurm et al. (2023) presented arithmetic primitives for efficient neuromorphic computing, and Mikaitis (2020) explored accelerators for transcendental functions in digital neuromorphic processors. Kwak et al. (2021) investigated precision requirements for floating-point arithmetic in SNNs.

However, these prior works typically focus on individual operations or use hybrid approaches combining neuromorphic and traditional computing elements. None provides a complete, bit-exact implementation of IEEE 754 arithmetic across all precisions (FP8/16/32/64) with full neural network layer support, nor establishes theoretical foundations for why such precision is achievable.

2.3. Neuron Models: LIF and GLIF

The Leaky Integrate-and-Fire (LIF) model is the most widely used spiking neuron model due to its computational simplicity and biological plausibility (Lu & Xu, 2022). Teeter et al. (2018) introduced the Generalized LIF (GLIF) framework, demonstrating that a family of models with increasing complexity can capture diverse neural behaviors. Yao et al. (2022) proposed a gated GLIF variant for deep SNNs, while Marasco et al. (2023) developed adaptive GLIF models for specific neuron types.

Our work extends the GLIF framework by proving that GLIF neurons with soft reset and dynamic thresholds are topologically equivalent to more complex Type II neurons (e.g., Hodgkin-Huxley), establishing that GLIF is not a simplification but a theoretically optimal choice for bit-exact computation.

2.4. Dynamical Systems and Topological Perspectives

The dynamical systems perspective on SNNs has received increasing attention. Zhang et al. (2021) analyzed spiking neural models through bifurcation theory, revealing fundamental dynamical properties. Wei et al. (2025) developed physics-informed SNNs for continuous-time systems. From a topological viewpoint, Papamarkou et al. (2024) advocated topological deep learning for understanding neural network structure, while Suresh et al. (2024) studied embedding space evolution using algebraic topology.

Our topological embedding theorem draws from these perspectives but focuses specifically on proving that GLIF networks create deterministic, chaos-free computation spaces homeomorphic to high-dimensional tori, providing theoretical guarantees for bit-exact arithmetic.

2.5. SNN Training Methods

Training SNNs is challenging due to the non-differentiability of spike generation. Surrogate gradient methods (Neftci et al., 2019; Zenke & Vogels, 2021) address this by replacing the discontinuous spike function with smooth surrogates during backpropagation. The Straight-Through Estimator (STE) is a particular surrogate that treats the gradient of the step function as identity (Yin et al., 2019; Chen et al., 2024).

Temporal backpropagation methods (Guo et al., 2023; Meng et al., 2023) propagate gradients through time steps but face memory and computational challenges. Our approach uses STE for training while preserving bit-exact forward computation, with a reserved interface for future temporal training modes.

2.6. Spiking Transformers and Attention

Recent work has adapted attention mechanisms and Transformer architectures to SNNs. Yao et al. (2023) proposed multi-dimensional attention for SNNs, demonstrating effectiveness across various tasks. Liu et al. (2022) developed hybrid top-down attention mechanisms. For full Transformer architectures, Yao et al. (2024) introduced Spike-driven Transformer V2 with a meta-architecture design, and Li et al. (2024) proposed Spikeformer achieving competitive performance with ANNs.

These works typically use rate-coded representations and approximate computations. In contrast, MofNeuroSim implements attention mechanisms with bit-exact IEEE 754 arithmetic, including Linear layers, LayerNorm/RMSNorm, Softmax, and Rotary Position Embeddings (RoPE).

2.7. Encoding and Normalization

Encoding continuous values into spike trains is fundamental to SNN computation. Date et al. (2023) proposed virtual neuron abstractions for encoding integers and rationals. For normalization, Layer Normalization (Ba et al., 2016) has become standard in Transformers, with Shao et al. (2020) analyzing its necessity in deep networks.

Our SAR-ADC inspired encoding with dynamic thresholds provides optimal information extraction per time step, while our normalization implementations (LayerNorm, RMSNorm) achieve bit-exact IEEE 754 compliance through pure SNN operations.

3. Methods

This section presents the mathematical foundations and SNN implementations of each component in MofNeuroSim. We first establish the theoretical basis for bit-exact computation using GLIF neurons, then organize the presentation hierarchically: from basic neuron models and logic gates, through arithmetic operations, to complete neural network layers.

3.1. Theoretical Foundations

This section establishes the mathematical foundations for bit-exact computation using Generalized Leaky Integrate-and-Fire (GLIF) networks. We prove that GLIF networks with soft reset and dynamic thresholds can construct differential topological embeddings, providing the theoretical basis for our bit-exact floating-point arithmetic.

3.1.1. Generalized LIF (GLIF) Neuron Model

The GLIF neuron generalizes the standard IF neuron by incorporating leakage and flexible reset mechanisms. The membrane potential dynamics is given by Equation 1:

$$V(t+1) = \beta V(t) + I(t) \tag{1}$$

where $V(t)$ denotes the membrane potential at time t, $\beta \in (0, 1]$ denotes the decay factor, and $I(t)$ denotes the input current.

The spike generation rule is given by Equation 2:

$$S(t) = H(V(t) - \theta(t)) \tag{2}$$

where $S(t)$ denotes the spike output (0 or 1), $\theta(t)$ denotes the firing threshold, and $H(\cdot)$ denotes the Heaviside step function.

Special Cases

- $\beta = 1$: Ideal IF neuron (no leakage) — used for bit-exact digital logic
- $\beta < 1$: Leaky IF (LIF) neuron — used for physical hardware simulation

Both cases support the theoretical framework when soft reset is employed.

3.1.2. Soft Reset: Topological Necessity

The reset mechanism after spike emission is critical for the system's topological properties.

Soft Reset The soft reset mechanism is defined by Equation 3:

$$V(t) \leftarrow V(t) - S(t) \cdot \theta(t) \tag{3}$$

where $V(t)$ denotes the membrane potential, $S(t) \in \{0, 1\}$ denotes the spike output, and $\theta(t)$ denotes the firing threshold. The membrane potential is reduced by exactly the threshold value upon spike emission, preserving any residual above threshold.

Hard Reset In contrast, the hard reset mechanism is defined by Equation 4:

$$V(t) \leftarrow 0 \quad (\text{if } S(t) = 1) \tag{4}$$

where the membrane potential is unconditionally reset to zero upon spike emission, discarding any residual information.

Lemma 3.1 (Toroidal Phase Space)**.** *Under soft reset dynamics, the GLIF network's phase space is diffeomorphic to an N-dimensional torus as given by Equation 5:*

$$\mathcal{M}_{GLIF} = \mathbb{T}^N \cong \underbrace{S^1 \times \cdots \times S^1}_{N} \tag{5}$$

where $\mathcal{M}_{GLIF}$ denotes the GLIF network phase space, $\mathbb{T}^N$ denotes the N-dimensional torus, S^1 denotes the unit circle, and N denotes the number of neurons.

Proof. The soft reset operation induces a modular arithmetic structure. Define the normalized state variable as given by Equation 6:

$$\phi = \frac{V}{\theta} \pmod 1 \tag{6}$$

where ϕ denotes the normalized phase variable, V denotes the membrane potential, and θ denotes the firing threshold. This maps $V \in \mathbb{R}$ to $\phi \in [0, 1) \cong S^1$ (the circle).

For N neurons with independent states $\phi_i \in S^1$, the total phase space is given by Equation 7:

$$\mathcal{M}_{GLIF} = \prod_{i=1}^{N} S^1 = \mathbb{T}^N \tag{7}$$

where $\mathcal{M}_{GLIF}$ denotes the GLIF network phase space, ϕ_i denotes the phase of neuron i, S^1 denotes the unit circle, N denotes the number of neurons, and $\mathbb{T}^N$ denotes the N-dimensional torus. This is a smooth, boundaryless compact Riemannian manifold.

Contrast with Hard Reset: Under hard reset ($V \leftarrow 0$), the mapping creates a discontinuity—the phase space becomes a manifold with boundary, violating the compactness requirements of embedding theorems. □

3.1.3. Dynamic Thresholds: Genericity and Reachability

Lemma 3.2 (Genericity via Dynamic Threshold)**.** *The deterministic threshold sequence $\{\theta_t = 2^{N-1-t}\}$ guarantees complete reachability of the discrete state space, which is the correct analog of genericity for bit-exact systems.*

Proof. We distinguish between continuous and discrete state spaces:

Continuous Systems (Classical Takens) For continuous manifolds, a constant threshold may cause phase-locking—trajectories confined to lower-dimensional periodic orbits. Time-varying thresholds (quasi-periodic or stochastic) break such degeneracies.

Bit-Exact Discrete Systems (MofNeuroSim) For finite-precision floating-point numbers $\mathbb{F}_p$ with $|\mathbb{F}_p| = 2^{32}$ (FP32), the state space is inherently discrete. Genericity transforms into *reachability*:

> *For any valid input $x \in \mathbb{F}_p$, the system must uniquely and reversibly map to a spike sequence* $\mathbf{s} \in \{0, 1\}^N$.

The SAR-ADC (Successive Approximation Register) threshold sequence $\theta_t = 2^{N-1-t}$ implements an information-theoretically optimal binary search:

- Each timestep extracts exactly 1 bit of information
- After N steps, the N-bit representation is fully determined
- No information loss, no redundancy

Define the encoding map $\Phi : \mathbb{F}_p \to \{0,1\}^N$ and decoding map $\Psi : \{0,1\}^N \to \mathbb{F}_p$ as given by Equation 8:

$$\begin{aligned} s_t &= \mathbb{I}(V_t \geq \theta_t) \\ V_{t+1} &= V_t - s_t \cdot \theta_t \\ \hat{x} = \Psi(\mathbf{s}) &= \sum_{t=0}^{N-1} s_t \cdot \theta_t \end{aligned} \tag{8}$$

where $s_t \in \{0,1\}$ denotes the spike at timestep t, $\mathbb{I}(\cdot)$ denotes the indicator function, V_t denotes the membrane potential at timestep t, θ_t denotes the threshold at timestep t, and $\hat{x}$ denotes the reconstructed value.

Claim: $\Psi \circ \Phi = \mathrm{id}_{\mathbb{F}_p}$ (bit-exact reconstruction).

Since $V_0 = |x|$ and soft reset preserves residuals, the final membrane potential is given by Equation 9:

$$V_N = V_0 - \sum_{t=0}^{N-1} s_t \cdot \theta_t = 0 \tag{9}$$

where V_N denotes the membrane potential after N timesteps, V_0 denotes the initial membrane potential, s_t denotes the spike at timestep t, and θ_t denotes the threshold at timestep t. Therefore $\sum_t s_t\theta_t = V_0 = |x|$, establishing bijectivity. $\square$

3.1.4. Observation Smoothness

Lemma 3.3 (Smoothness of Observation Function)**.** *Under the bit-exact premise, the binary coding interpretation function $h : \mathbb{T}^N \to \mathbb{R}$ is smooth (or piecewise smooth), satisfying embedding theorem requirements.*

Proof. Let $\mathbf{s} \in \{0,1\}^{\mathbb{Z}}$ be the binary spike train. The coding interpretation is given by Equation 10:

$$y_t = h(\boldsymbol{\phi}_t) = \sum_{k=0}^{K} w_k \cdot \mathbb{I}(\text{Spike at } t-k) \tag{10}$$

where y_t denotes the observation at time t, $h(\cdot)$ denotes the observation function, $\boldsymbol{\phi}_t$ denotes the phase state at time t, K denotes the number of delay taps, $w_k = 2^{N-1-k}$ denotes the coding weight for delay k, and $\mathbb{I}(\cdot)$ denotes the indicator function.

The spike generation involves a Heaviside step function, which is discontinuous at $V_t = \theta_t$. However:

1. **Soft reset continuity**: Under soft reset, the spike timing varies continuously with small perturbations in input.
2. **Discrete grid resolution**: For bit-exact systems, inputs lie on a discrete floating-point grid. The discontinuity is "absorbed" by quantization—within each quantization cell, the function is constant (hence smooth).
3. **Multi-neuron averaging**: For networks, individual discontinuities average out, yielding effectively smooth macroscopic observations.

Therefore, the weighted sum y_t depends continuously on the internal state $\boldsymbol{\phi}_t$. $\square$

3.1.5. Main Theorem: GLIF Topological Embedding

Theorem 3.4 (GLIF Topological Embedding)**.** *If the GLIF network dimension N and delay embedding dimension m satisfy $m > 2d_{\mathcal{A}}$ (where $d_{\mathcal{A}}$ is the attractor dimension), then the observation sequence of the GLIF network suffices to construct a differential topological embedding of the attractor $\mathcal{A}$.*

Proof. By the Generalized Takens Embedding Theorem, we verify:

1. **Manifold**: $\mathcal{M}_{GLIF}$ is smooth and compact (Lemma 3.1)
2. **Dynamics**: The evolution map is a diffeomorphism with full reachability (Lemma 3.2)
3. **Observation**: The observation function h is smooth/piecewise smooth (Lemma 3.3)
4. **Dimension**: $m > 2d_{\mathcal{A}}$ is satisfied for sufficient N

Therefore, the delay map given by Equation 11 is a differential topological embedding:

$$\Psi_h(\mathbf{y}) = [y_t, y_{t-\tau}, y_{t-2\tau}, \ldots, y_{t-(m-1)\tau}] \tag{11}$$

where Ψ_h denotes the delay embedding map, $\mathbf{y}$ denotes the observation sequence, y_t denotes the observation at time t, τ denotes the delay interval, and m denotes the embedding dimension. $\square$

3.1.6. Significance of the Theorem

Theorem 3.4 is not merely a mathematical curiosity but provides three profound implications that elevate this work from an engineering achievement to a theoretical breakthrough:

Paradigm Shift The theorem proves that GLIF-based SNN computation spaces are *deterministic and chaos-free*, with phase spaces homeomorphic to high-dimensional tori. This fundamentally overturns the traditional perception that "SNNs are inherently fuzzy," **theoretically establishing the possibility of exact computation in spiking neural networks**.

Table 1. Soft Reset Throughout MofNeuroSim

Component	Reason
Encoder	Preserve residual
Logic Gates	Toroidal topology
Arithmetic	Consistent dynamics
Decoder	Boundary (no neurons)

Universality of the GLIF Model A common concern is whether choosing the simple GLIF model sacrifices biological realism compared to detailed Hodgkin-Huxley (HH) neurons. Our theorem proves that GLIF neurons are *topologically equivalent* to Type II neurons like HH in computational capability—both can construct differential embeddings of equivalent dimension. Therefore, choosing GLIF is **not a compromise for simplicity, but a theoretically optimal choice** under rigorous mathematical guidance.

Unified Framework The theorem applies uniformly to ideal IF neurons ($\beta = 1$) and physically realistic LIF neurons ($\beta < 1$). This paves the way for **unifying exact computation (IF) with biological dynamics and hardware physics (LIF)** within the same mathematical framework, enabling seamless transition from algorithmic development to physical deployment.

3.1.7. Implications for MofNeuroSim

Theorem 3.4 provides the theoretical justification for our architecture:

1. **Soft reset is mandatory**: Hard reset breaks the toroidal topology, preventing bit-exact encoding. The encoder *must* use soft reset.
2. **Dynamic thresholds enable bit extraction**: The SAR-ADC sequence $\theta_t = 2^{N-1-t}$ is not arbitrary but information-theoretically optimal for binary encoding.
3. **IF and LIF are unified**: Both $\beta = 1$ (IF) and $\beta < 1$ (LIF) preserve the topological structure under soft reset. This enables:
    - IF neurons for bit-exact digital logic (computation paths)
    - LIF neurons for physical hardware simulation (robustness testing)
4. **Soft reset is universal**: All neurons throughout MofNeuroSim use soft reset to maintain the toroidal phase space topology. This ensures theoretical consistency from encoders through logic gates to arithmetic units.

3.2. Architectural Overview

MofNeuroSim implements IEEE 754 floating-point arithmetic entirely through spiking neural networks (SNNs) using Generalized Leaky Integrate-and-Fire (GLIF) neurons. As established in Section 3.1, GLIF networks with soft reset and dynamic thresholds provide the theoretical foundations for bit-exact computation. All computations are performed through spike-based logic gates constructed from GLIF neurons.

3.2.1. Computation Flow

The general computation flow follows a three-stage pipeline as given by Equation 12:

$$\text{Float} \xrightarrow{\text{Encoder}} \text{Pulse} \xrightarrow{\text{SNN Gates}} \text{Pulse} \xrightarrow{\text{Decoder}} \text{Float} \qquad (12)$$

where Float denotes the floating-point domain, Pulse denotes the binary pulse domain, Encoder denotes the boundary operation converting floats to pulses, SNN Gates denotes the spike-based computation, and Decoder denotes the boundary operation converting pulses back to floats. The encoder and decoder serve as **boundary operations** between the continuous floating-point domain and the discrete pulse domain. Within the pulse domain, all operations are performed using SNN gates that process binary spike sequences.

3.2.2. Hierarchical Architecture

The system is organized into five hierarchical levels, each building upon the previous:

- **Level 0 – Neurons**: Generalized Leaky Integrate-and-Fire (GLIF) neurons providing the computational primitives. With decay factor $\beta = 1$, GLIF reduces to ideal IF neurons for bit-exact digital logic; with $\beta < 1$, it becomes LIF for physical hardware simulation. Both variants use soft reset to preserve the toroidal phase space topology (Theorem 3.4).
- **Level 1 – Logic Gates**: Boolean gates (AND, OR, NOT, XOR, MUX) constructed from GLIF neurons with fixed thresholds. These gates process binary spike values.
- **Level 2 – Arithmetic Units**: Multi-bit adders, subtractors, comparators, and shifters built from Level 1 gates. These implement integer arithmetic on pulse sequences.
- **Level 3 – Floating-Point Operators**: IEEE 754 compliant operators (adder, multiplier, divider, square root) that process sign, exponent, and mantissa fields separately using Level 2 units.

Table 2. Notation and Symbols

Symbol	Description
$\mathbf{P}_x$	Pulse representation of value x
$\mathbf{s}$	Single spike (binary: 0 or 1)
V	Membrane potential
θ	Firing threshold
$H(\cdot)$	Heaviside step function
s, e, m	Sign, exponent, mantissa bits
n_e, n_m	Number of exponent/mantissa bits
bias	Exponent bias ($2^{n_e-1} - 1$)
β	LIF decay factor ($0 < \beta < 1$)

Table 3. Supported Floating-Point Formats

Format	Bits	Sign	Exp	Mant
FP8 (E4M3)	8	1	4	3
FP16	16	1	5	10
FP32	32	1	8	23
FP64	64	1	11	52

- **Level 4 – Neural Network Layers**: Complete layers (Linear, LayerNorm, RMSNorm, Attention) and activation functions (Sigmoid, Tanh, GELU, Softmax) built from Level 3 operators.

3.2.3. Notation and Symbols

Table 2 summarizes the notation used throughout this paper.

3.2.4. Supported Precisions

MofNeuroSim supports multiple IEEE 754 floating-point formats, each achieving bit-exact results:

3.3. Pulse Encoding

This section describes how IEEE 754 floating-point numbers are encoded as binary pulse sequences using dynamic threshold GLIF neurons. The theoretical foundations for this encoding—including why soft reset is mandatory—are established in Section 3.1.

3.3.1. IEEE 754 Floating-Point Representation

A floating-point number x in IEEE 754 format is represented as given by Equation 13:

$$x = (-1)^s \times 2^{e-\text{bias}} \times (1.m) \tag{13}$$

where $s \in \{0, 1\}$ denotes the sign bit, e denotes the unsigned exponent, bias $= 2^{n_e-1} - 1$ denotes the exponent bias (with n_e being the number of exponent bits), and m denotes the fractional mantissa with an implicit leading 1 for normalized numbers.

For FP32, the 32-bit representation is structured as shown in Equation 14:

$$\underbrace{s}_{1}\underbrace{e_7e_6\cdots e_0}_{8}\underbrace{m_{22}m_{21}\cdots m_0}_{23} \tag{14}$$

where s denotes the 1-bit sign field, $e_7e_6\cdots e_0$ denotes the 8-bit exponent field, and $m_{22}m_{21}\cdots m_0$ denotes the 23-bit mantissa field.

3.3.2. Pulse Sequence Representation

A floating-point number is encoded as a pulse sequence $\mathbf{P}_x = [p_0, p_1, \ldots, p_{n-1}]$, where each pulse $p_i \in \{0, 1\}$ corresponds to the i-th bit of the IEEE 754 representation as given by Equation 15:

$$\mathbf{P}_x = [s, e_{n_e-1}, \ldots, e_0, m_{n_m-1}, \ldots, m_0] \tag{15}$$

where s denotes the sign bit, $e_{n_e-1}, \ldots, e_0$ denote the exponent bits (from most to least significant), and $m_{n_m-1}, \ldots, m_0$ denote the mantissa bits. For FP32, this yields a 32-element binary vector. The pulse values are represented as floating-point tensors containing only 0.0 or 1.0 values.

3.3.3. Dynamic Threshold GLIF Neuron

The encoder uses a **dynamic threshold GLIF neuron** to extract bits from floating-point values. For bit-exact encoding, we use $\beta = 1$ (ideal IF mode). The dynamic threshold decreases by powers of 2 to extract successive bits via the SAR-ADC (Successive Approximation Register) algorithm.

Dynamics The membrane potential $V(t)$ evolves according to Equation 16:

$$V(t + 1) = \beta V(t) + I(t) \tag{16}$$

where $V(t)$ denotes the membrane potential at time t, β denotes the decay factor ($\beta = 1$ for bit-exact mode), and $I(t)$ denotes the input current. The neuron fires when the condition in Equation 17 is satisfied:

$$\mathbf{s}(t) = H(V(t) - \theta(t)) \tag{17}$$

where $\mathbf{s}(t) \in \{0, 1\}$ denotes the spike output, $\theta(t)$ denotes the dynamic threshold, and $H(\cdot)$ denotes the Heaviside step function.

Dynamic Threshold Schedule As established in Lemma 3.2, the threshold at timestep t follows the information-theoretically optimal SAR sequence given by Equation 18:

$$\theta(t) = 2^{n-1-t} \tag{18}$$

where n denotes the total number of bits to extract, and $t \in \{0, 1, \ldots, n-1\}$ denotes the timestep. This schedule extracts exactly 1 bit of information per timestep, from most significant to least significant.

Soft Reset After firing, the membrane potential is updated according to the soft reset rule given by Equation 19:

$$V(t+1) = V(t) - \mathbf{s}(t) \cdot \theta(t) \tag{19}$$

where $V(t)$ denotes the current membrane potential, $\mathbf{s}(t)$ denotes the spike output (0 or 1), and $\theta(t)$ denotes the current threshold. The soft reset is **mandatory** for the encoder (Lemma 3.1): it preserves the residual membrane potential, allowing subsequent bits to be extracted correctly. A hard reset ($V \leftarrow 0$) would destroy the toroidal phase space topology and lose information.

3.3.4. Encoding Algorithm

For a positive floating-point value v with n-bit representation:

1. Initialize: $V(0) = v$
2. For $t = 0, 1, \ldots, n-1$:
   (a) Compute threshold: $\theta(t) = 2^{n-1-t}$
   (b) Generate spike: $\mathbf{s}(t) = H(V(t) - \theta(t))$
   (c) Update membrane: $V(t+1) = V(t) - \mathbf{s}(t) \cdot \theta(t)$
3. Output: $\mathbf{P}_v = [\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(n-1)]$

Example Encoding the value 5 with 4 bits:

$$t = 0: \quad \theta = 8, \quad V = 5 < 8, \quad \mathbf{s} = 0 \tag{20}$$
$$t = 1: \quad \theta = 4, \quad V = 5 \geq 4, \quad \mathbf{s} = 1, \quad V \leftarrow 1 \tag{21}$$
$$t = 2: \quad \theta = 2, \quad V = 1 < 2, \quad \mathbf{s} = 0 \tag{22}$$
$$t = 3: \quad \theta = 1, \quad V = 1 \geq 1, \quad \mathbf{s} = 1 \tag{23}$$

Result: $\mathbf{P}_5 = [0, 1, 0, 1]_2 = 5_{10}$ ✓

3.3.5. Sign Bit Handling

For floating-point numbers, the sign bit requires special handling. The sign bit is extracted as given by Equation 24:

$$s = H(-x) \tag{24}$$

where $s \in \{0, 1\}$ denotes the sign bit, x denotes the input floating-point value, and $H(\cdot)$ denotes the Heaviside step function. The absolute value $|x|$ is then encoded for the exponent and mantissa fields using the standard dynamic threshold mechanism. The sign bit is prepended to form the complete pulse sequence.

3.3.6. Decoding

The decoder reconstructs the floating-point value from the pulse sequence. This is a **boundary operation** where traditional arithmetic is permitted. The decoding formula is given by Equation 25:

$$x = (-1)^{p_0} \cdot 2^{E - \text{bias}} \cdot (1 + M) \tag{25}$$

where $E = \sum_{i=1}^{n_e} p_i \cdot 2^{n_e - i}$ is the exponent and $M = \sum_{j=1}^{n_m} p_{n_e+j} \cdot 2^{-j}$ is the mantissa fraction. where p_0 denotes the sign bit, $p_1, \ldots, p_{n_e}$ denote the exponent bits, $p_{n_e+1}, \ldots, p_{n_e+n_m}$ denote the mantissa bits, n_e denotes the number of exponent bits, n_m denotes the number of mantissa bits, and bias denotes the exponent bias. The decoder operates at the system boundary, converting pulse sequences back to floating-point for output.

3.4. SNN Logic Gates

This section presents the construction of Boolean logic gates using GLIF neurons. Each gate is implemented by choosing an appropriate threshold that produces the desired truth table.

3.4.1. GLIF Neuron Model for Logic Gates

For logic gates, we use GLIF neurons with $\beta = 1$ (ideal IF mode), fixed threshold θ, and soft reset. Consistent with the theoretical framework (Section 3.1), soft reset is used throughout MofNeuroSim to maintain the toroidal phase space topology. The general gate output is given by Equation 26:

$$y = H\left(\sum_i w_i x_i - \theta\right) \tag{26}$$

where $y \in \{0, 1\}$ denotes the gate output, $x_i \in \{0, 1\}$ denote the binary inputs, w_i denote the synaptic weights, θ denotes the firing threshold, and $H(\cdot)$ denotes the Heaviside step function defined by Equation 27:

$$H(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{27}$$

where z denotes the input argument.

3.4.2. AND Gate

The AND gate outputs 1 only when all inputs are 1. For two inputs with unit weights, the AND gate is defined by Equation 28:

$$\text{AND}(x_1, x_2) = H(x_1 + x_2 - 1.5) \tag{28}$$

where $x_1, x_2 \in \{0, 1\}$ denote the binary inputs, and the threshold $\theta = 1.5$ ensures the neuron fires only when both inputs are active.

Verification

$$\text{AND}(0,0) = H(-1.5) = 0 \tag{29}$$
$$\text{AND}(0,1) = H(-0.5) = 0 \tag{30}$$
$$\text{AND}(1,0) = H(-0.5) = 0 \tag{31}$$
$$\text{AND}(1,1) = H(+0.5) = 1 \tag{32}$$

3.4.3. OR Gate

The OR gate outputs 1 when at least one input is 1. The OR gate is defined by Equation 33:

$$\text{OR}(x_1, x_2) = H(x_1 + x_2 - 0.5) \tag{33}$$

where $x_1, x_2 \in \{0, 1\}$ denote the binary inputs, and the threshold $\theta = 0.5$ ensures the neuron fires when at least one input is active.

Verification

$$\text{OR}(0,0) = H(-0.5) = 0 \tag{34}$$
$$\text{OR}(0,1) = H(+0.5) = 1 \tag{35}$$
$$\text{OR}(1,0) = H(+0.5) = 1 \tag{36}$$
$$\text{OR}(1,1) = H(+1.5) = 1 \tag{37}$$

3.4.4. NOT Gate

The NOT gate inverts the input using an inhibitory connection. The NOT gate is defined by Equation 38:

$$\text{NOT}(x) = H(1 - x - 0.5) = H(0.5 - x) \tag{38}$$

where $x \in \{0, 1\}$ denotes the binary input. This can be interpreted as a neuron with a constant bias of 1, an inhibitory weight of -1, and threshold 0.5.

Verification

$$\text{NOT}(0) = H(0.5) = 1 \tag{39}$$
$$\text{NOT}(1) = H(-0.5) = 0 \tag{40}$$

3.4.5. XOR Gate

The XOR gate cannot be implemented with a single neuron (it is not linearly separable). We implement it using a five-neuron network as given by Equation 41:

$$\text{XOR}(a, b) = (a \wedge \neg b) \vee (\neg a \wedge b) \tag{41}$$

where $a, b \in \{0, 1\}$ denote the binary inputs. This requires 2 NOT gates, 2 AND gates, and 1 OR gate.

Verification

$$\text{XOR}(0,0) = (0 \wedge 1) \vee (1 \wedge 0) = 0 \vee 0 = 0 \tag{42}$$
$$\text{XOR}(0,1) = (0 \wedge 0) \vee (1 \wedge 1) = 0 \vee 1 = 1 \tag{43}$$
$$\text{XOR}(1,0) = (1 \wedge 1) \vee (0 \wedge 0) = 1 \vee 0 = 1 \tag{44}$$
$$\text{XOR}(1,1) = (1 \wedge 0) \vee (0 \wedge 1) = 0 \vee 0 = 0 \tag{45}$$

3.4.6. Multiplexer (MUX) Gate

The 2-to-1 multiplexer selects between two inputs based on a select signal. The MUX gate is defined by Equation 46:

$$\text{MUX}(s, a, b) = \text{OR}(\text{AND}(s, a), \text{AND}(\text{NOT}(s), b)) \tag{46}$$

where $s \in \{0, 1\}$ denotes the select signal and $a, b \in \{0, 1\}$ denote the data inputs. When $s = 1$, the output is a; when $s = 0$, the output is b.

3.4.7. Half Adder

The half adder computes the sum and carry of two single-bit inputs as given by Equation 48:

$$\text{sum} = \text{XOR}(a, b) \tag{47}$$
$$\text{carry} = \text{AND}(a, b) \tag{48}$$

where $a, b \in \{0, 1\}$ denote the binary inputs, sum denotes the sum bit, and carry denotes the carry-out bit.

3.4.8. Full Adder

The full adder extends the half adder to include a carry input. The full adder is defined by Equation 50:

$$\text{sum} = \text{XOR}(\text{XOR}(a, b), c_{in}) \tag{49}$$
$$\text{carry} = \text{OR}(\text{AND}(a, b), \text{AND}(\text{XOR}(a, b), c_{in})) \tag{50}$$

where $a, b \in \{0, 1\}$ denote the binary inputs, $c_{in} \in \{0, 1\}$ denotes the carry-in, sum denotes the sum output, and carry denotes the carry-out. This produces:

$$a + b + c_{in} = 2 \cdot \text{carry} + \text{sum} \tag{51}$$

3.4.9. Multi-bit Adder

An n-bit ripple-carry adder chains n full adders as given by Equation 52:

$$(\text{sum}_i, c_{i+1}) = \text{FullAdder}(a_i, b_i, c_i) \tag{52}$$

where a_i, b_i denote the i-th bits of the input operands, c_i denotes the carry-in for bit i, sum_i denotes the i-th sum bit, and c_{i+1} denotes the carry-out. The index i ranges from 0 to $n - 1$, with $c_0 = 0$ for addition or $c_0 = 1$ for subtraction via two's complement.

Table 4. SNN Logic Gate Implementations

Gate	θ	**Formula**	**N**
AND	1.5	$H(x_1 + x_2 - 1.5)$	1
OR	0.5	$H(x_1 + x_2 - 0.5)$	1
NOT	0.5	$H(0.5 - x)$	1
XOR	–	OR(AND, AND)	5
MUX	–	OR(AND, AND)	4

3.4.10. Comparator

Comparison of two n-bit numbers uses cascaded single-bit comparators. For each bit position i (from MSB to LSB), the comparison signals are computed as given by Equation 53:

$$\begin{aligned} \text{gt}_i &= \text{AND}(a_i, \text{NOT}(b_i)) \\ \text{lt}_i &= \text{AND}(\text{NOT}(a_i), b_i) \\ \text{eq}_i &= \text{NOT}(\text{XOR}(a_i, b_i)) \end{aligned} \tag{53}$$

where $a_i, b_i \in \{0, 1\}$ denote the i-th bits of the two operands, gt_i denotes the greater-than signal, lt_i denotes the less-than signal, and eq_i denotes the equal signal. The final comparison result propagates from the most significant differing bit.

3.4.11. Gate Summary

Table 4 summarizes the logic gate implementations:

3.4.12. Vectorized Gates

For efficient batch processing, all gates support vectorized operations on pulse tensors. Given input tensors $\mathbf{A}, \mathbf{B} \in \{0, 1\}^{B \times N}$ (batch size B, N bits), the vectorized gate applies element-wise as given by Equation 54:

$$\mathbf{Y}_{b,n} = \text{Gate}(\mathbf{A}_{b,n}, \mathbf{B}_{b,n}) \tag{54}$$

where $\mathbf{Y}_{b,n}$ denotes the output at batch index b and bit position n, $\mathbf{A}_{b,n}$ and $\mathbf{B}_{b,n}$ denote the corresponding input elements, and $\text{Gate}(\cdot, \cdot)$ denotes any binary gate operation. This enables parallel processing of multiple floating-point values and their bit representations.

3.5. Floating-Point Arithmetic Operations

This section describes how IEEE 754 floating-point arithmetic operations are implemented using SNN logic gates. Each operation processes the sign, exponent, and mantissa fields separately using the gates defined in Section 3.4.

3.5.1. Floating-Point Addition

Floating-point addition is the most complex arithmetic operation, requiring exponent alignment, mantissa addition, and normalization.

Algorithm Overview Given two floating-point numbers $a = (-1)^{s_a} \times 2^{e_a} \times m_a$ and $b = (-1)^{s_b} \times 2^{e_b} \times m_b$, the computation proceeds as: compare exponents ($d = e_a - e_b$) using SNN subtractor, align mantissas by right-shifting the smaller by $|d|$ bits, add or subtract aligned mantissas based on signs, normalize result to restore $1.xxx$ format while adjusting exponent, then apply IEEE 754 rounding.

Exponent Difference The exponent difference is computed using an n_e-bit SNN subtractor as given by Equation 55:

$$d = e_a - e_b = e_a + \overline{e_b} + 1 \tag{55}$$

where d denotes the exponent difference, e_a, e_b denote the exponents of the two operands, and $\overline{e_b}$ denotes the bitwise NOT (one's complement) of e_b. Adding 1 completes the two's complement subtraction.

Mantissa Alignment The mantissa of the smaller number is right-shifted by $|d|$ positions using a barrel shifter constructed from MUX gates. For each shift amount 2^k, the shifted mantissa bit is computed as given by Equation 56:

$$m'_i = \text{MUX}(m_i, m_{i+2^k}, d_k) \tag{56}$$

where m'_i denotes the i-th bit of the shifted mantissa, m_i denotes the original mantissa bit, and d_k denotes the k-th bit of the shift amount controlling whether to shift by 2^k positions.

Mantissa Addition/Subtraction The operation type is determined by the sign bits as given by Equation 57:

$$\text{op} = \text{XOR}(s_a, s_b) \tag{57}$$

where s_a, s_b denote the sign bits of the two operands, and op denotes the operation flag. If $\text{op} = 0$ (same signs), add mantissas; if $\text{op} = 1$ (different signs), subtract.

Normalization The result mantissa may require normalization: for overflow ($m \geq 2$), right-shift by 1 and increment exponent; for underflow ($m < 1$), count leading zeros, left-shift, and decrement exponent. Leading zero count (LZC) is implemented using a priority encoder built from OR gates.

3.5.2. Floating-Point Multiplication

Multiplication is structurally simpler than addition. The product is computed as given by Equation 58:

$$a \times b = (-1)^{s_a \oplus s_b} \times 2^{(e_a + e_b - \text{bias})} \times (m_a \times m_b) \tag{58}$$

where s_a, s_b denote the sign bits, e_a, e_b denote the exponents, m_a, m_b denote the mantissas, and bias denotes the exponent bias.

Sign Computation The result sign is computed using XOR as given by Equation 59:

$$s_r = \text{XOR}(s_a, s_b) \tag{59}$$

where s_r denotes the result sign bit.

Exponent Addition The result exponent is computed as given by Equation 60:

$$e_r = e_a + e_b - \text{bias} \tag{60}$$

where e_r denotes the result exponent. The bias subtraction prevents double-counting. This is implemented using SNN adders:

$$e_r = \text{Adder}(e_a, e_b) - \text{bias} \tag{61}$$

Mantissa Multiplication For mantissas $m_a = 1.a_{n_m-1} \dots a_0$ and $m_b = 1.b_{n_m-1} \dots b_0$, the product is computed using an array of AND gates and adders as given by Equation 62:

$$m_a \times m_b = \sum_{i=0}^{n_m} \sum_{j=0}^{n_m} a_i \cdot b_j \cdot 2^{-(i+j)} \tag{62}$$

where a_i, b_j denote the individual mantissa bits and n_m denotes the number of mantissa bits. Each partial product $a_i \cdot b_j$ is computed with an AND gate. The partial products are accumulated using a tree of SNN adders.

Normalization Since $1 \leq m_a, m_b < 2$, the product satisfies $1 \leq m_a \times m_b < 4$. If $m_r \geq 2$, the result is normalized as given by Equation 63:

$$\begin{aligned} m_r &\leftarrow m_r/2 \quad \text{(right-shift by 1)} \\ e_r &\leftarrow e_r + 1 \end{aligned} \tag{63}$$

where m_r denotes the result mantissa and e_r denotes the result exponent.

3.5.3. Floating-Point Division

Division uses the Newton-Raphson method to compute the reciprocal as given by Equation 64:

$$\frac{a}{b} = a \times \frac{1}{b} \tag{64}$$

where the reciprocal $1/b$ is computed iteratively.

Newton-Raphson Reciprocal To find $x = 1/b$, we iterate using the formula given by Equation 65:

$$x_{n+1} = x_n(2 - b \cdot x_n) \tag{65}$$

where x_n denotes the n-th approximation to the reciprocal and b denotes the divisor mantissa. This converges quadratically. With a good initial estimate, 3-4 iterations achieve FP32 precision.

Initial Estimate The initial estimate uses a lookup table indexed by the high bits of the mantissa as given by Equation 66:

$$x_0 \approx \frac{1}{m_b} \tag{66}$$

where m_b denotes the mantissa of the divisor. The table is precomputed and stored in SNN-compatible format.

Complete Division The computation proceeds as: compute sign ($s_r = \text{XOR}(s_a, s_b)$), compute exponent ($e_r = e_a - e_b + \text{bias}$), compute reciprocal of m_b using Newton-Raphson, multiply ($m_r = m_a \times (1/m_b)$), then normalize the result.

3.5.4. Floating-Point Square Root

Square root also uses Newton-Raphson iteration as given by Equation 67:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{67}$$

where x_n denotes the n-th approximation to $\sqrt{a}$ and a denotes the input value.

Exponent Handling For $a = 2^e \times m$, the square root decomposes as given by Equation 68:

$$\sqrt{a} = \sqrt{2^e \times m} = 2^{e/2} \times \sqrt{m} \tag{68}$$

where e denotes the exponent and m denotes the mantissa. When e is odd, the formula becomes:

$$\sqrt{a} = 2^{(e-1)/2} \times \sqrt{2m} \tag{69}$$

Initial Estimate The initial estimate for $\sqrt{m}$ (where $1 \leq m < 2$) is computed as given by Equation 70:

$$x_0 = 0.5 + 0.5 \times m \tag{70}$$

where m denotes the mantissa value. This linear approximation provides a starting point within 25% of the true value.

Complete Square Root The computation proceeds as: check sign (if $s = 1$, return NaN), decompose exponent ($e = 2q + r$ where $r \in \{0, 1\}$), adjust mantissa ($m' = m \times 2^r$), compute $\sqrt{m'}$ using Newton-Raphson, then set result exponent ($e_r = q + \text{bias}$).

3.5.5. Special Value Handling

All arithmetic operations handle IEEE 754 special values: zero (exponent and mantissa all zeros), infinity (exponent all ones, mantissa all zeros), NaN (exponent all ones, mantissa non-zero), and denormals (exponent all zeros, mantissa non-zero). Special value detection uses OR-reduction gates on exponent/mantissa fields as given by Equation 71:

$$\begin{aligned} \text{exp\_zero} &= \text{NOR}(e_{n_e-1}, \ldots, e_0) \\ \text{exp\_max} &= \text{AND}(e_{n_e-1}, \ldots, e_0) \\ \text{mant\_zero} &= \text{NOR}(m_{n_m-1}, \ldots, m_0) \end{aligned} \tag{71}$$

where $e_{n_e-1}, \ldots, e_0$ denote the exponent bits, $m_{n_m-1}, \ldots, m_0$ denote the mantissa bits, exp_zero indicates all-zero exponent, exp_max indicates all-one exponent, and mant_zero indicates all-zero mantissa.

3.6. Activation Functions

This section describes how common neural network activation functions are implemented using SNN arithmetic operators. Each function is decomposed into floating-point operations from Section 3.5.

3.6.1. Exponential Function

The exponential function e^x is fundamental to many activations. We implement it using a table-driven algorithm entirely through SNN operators.

Algorithm Overview The computation uses $N = 32$ subdivisions per octave as given by Equation 72:

$$z = \text{round}(x \cdot N/\ln 2), \quad k = \lfloor z/N \rfloor, \quad j = z \mod N \tag{72}$$

where z denotes the scaled input, k denotes the integer exponent, and $j \in [0, 31]$ denotes the table index.

SNN Implementation The computation proceeds entirely through SNN gates:

- **Scaling**: $x \cdot (N/\ln 2)$ via SNN multiplier with precomputed constant
- **Rounding**: SNN floor function using comparators and MUX gates
- **Table lookup**: 5-layer MUX tree selects from 32 precomputed $T[j] = 2^{j/32}$ constants stored as pulse sequences
- **Remainder**: $r = x - z \cdot \ln 2$ via SNN subtraction

Polynomial Evaluation A degree-3 minimax polynomial approximates e^r for $|r| < \ln 2/64$:

$$P(r) = 1 + r + C_2 r^2 + C_3 r^3 \tag{73}$$

where C_2, C_3 are precomputed pulse constants. Horner's method ($P = 1 + r(1 + r(C_2 + r \cdot C_3))$) minimizes SNN multiplications.

Result Reconstruction The final result combines components using SNN operators:

$$e^x = 2^k \times T[j] \times P(r) \tag{74}$$

where 2^k scaling adjusts the exponent field directly via SNN bit manipulation, and $T[j] \times P(r)$ uses SNN multiplication.

3.6.2. Sigmoid Function

The sigmoid activation is defined by Equation 75:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{75}$$

where x denotes the input value and $\sigma(x) \in (0, 1)$ denotes the output.

SNN Implementation The computation proceeds as: negate ($x \to -x$), exponential (e^{-x}), add one ($1 + e^{-x}$), then reciprocal via Newton-Raphson to obtain $1/(1 + e^{-x})$.

Symmetry Property For numerical stability with large positive inputs:

$$\sigma(x) = 1 - \sigma(-x) \tag{76}$$

When $x > 0$, we compute $\sigma(-x)$ to avoid overflow in e^{-x}.

3.6.3. Hyperbolic Tangent

The tanh activation is related to sigmoid as given by Equation 77:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} = 2\sigma(2x) - 1 \tag{77}$$

where x denotes the input value, $\sigma(\cdot)$ denotes the sigmoid function, and $\tanh(x) \in (-1, 1)$ denotes the output.

SNN Implementation Using the sigmoid-based form: scale ($x \to 2x$), apply sigmoid ($\sigma(2x)$), scale again ($2\sigma(2x)$), then subtract one to obtain $2\sigma(2x) - 1$.

3.6.4. GELU (Gaussian Error Linear Unit)

GELU is defined by Equation 78:

$$\text{GELU}(x) = x \cdot \Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{78}$$

where x denotes the input value, $\Phi(x)$ denotes the Gaussian CDF, and $\text{erf}(\cdot)$ denotes the error function.

Fast Approximation We use the sigmoid-based approximation given by Equation 79:

$$\text{GELU}(x) \approx x \cdot \sigma(1.702\,x) \tag{79}$$

where $\sigma(\cdot)$ denotes the sigmoid function and 1.702 is an empirically determined constant. This approximation has maximum error < 0.005.

SNN Implementation The computation proceeds as: scale ($x \to 1.702x$), apply sigmoid ($\sigma(1.702x)$), then multiply by x to obtain $x \cdot \sigma(1.702x)$.

Exact GELU For applications requiring higher precision, we implement the exact formula using the error function Taylor series given by Equation 80:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{N} \frac{(-1)^n x^{2n+1}}{n!(2n+1)} \tag{80}$$

where $\text{erf}(x)$ denotes the error function, x denotes the input value, and N denotes the number of Taylor series terms.

3.6.5. SiLU (Sigmoid Linear Unit)

SiLU, also known as Swish, is defined by Equation 81:

$$\text{SiLU}(x) = x \cdot \sigma(x) \tag{81}$$

where x denotes the input value and $\sigma(\cdot)$ denotes the sigmoid function.

SNN Implementation The computation proceeds as: apply sigmoid ($\sigma(x)$), then multiply by x to obtain $x \cdot \sigma(x)$.

3.6.6. ReLU (Rectified Linear Unit)

ReLU is the simplest activation as defined by Equation 82:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{82}$$

Table 5. Activation Function Complexity (FP32)

Activation	Operations	Cost
ReLU	1 NOT, 32 AND	1×
Sigmoid	exp, div	15×
Tanh	sigmoid, mul, sub	18×
GELU (fast)	sigmoid, 2 mul	20×
SiLU	sigmoid, mul	17×
Softmax	exp, div	15×

where x denotes the input value.

SNN Implementation ReLU is implemented by checking the sign bit as given by Equation 83:

$$\text{ReLU}(\mathbf{P}_x) = \text{AND}(\mathbf{P}_x, \text{NOT}(s_x)) \tag{83}$$

where $\mathbf{P}_x$ denotes the pulse representation of x and s_x denotes the sign bit (pulse 0 of $\mathbf{P}_x$). When $s_x = 1$ (negative), all bits are zeroed.

This requires only a single AND gate per bit plus one NOT gate for the sign, making it extremely efficient.

3.6.7. Softmax

Softmax converts a vector of logits to probabilities as given by Equation 84:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{84}$$

where x_i denotes the i-th input logit, n denotes the number of elements, and the output represents a probability distribution that sums to 1.

Numerical Stability For stability, we subtract the maximum value as given by Equation 85:

$$\text{Softmax}(x_i) = \frac{e^{x_i - x_{\max}}}{\sum_{j=1}^{n} e^{x_j - x_{\max}}} \tag{85}$$

where $x_{\max} = \max_j(x_j)$ denotes the maximum input value. This prevents overflow in the exponential computation.

SNN Implementation The computation proceeds as: find maximum ($x_{\max}$) using SNN comparators, subtract ($x'_i = x_i - x_{\max}$), compute exponentials ($e_i = e^{x'_i}$), sum ($S = \sum_j e_j$) using an adder tree, then normalize ($p_i = e_i/S$) using SNN division.

3.6.8. Complexity Analysis

Table 5 summarizes the computational complexity of each activation:

3.7. Linear Layer

This section describes the implementation of fully-connected (linear) layers using SNN arithmetic operators. The linear layer is the fundamental building block of neural networks.

3.7.1. Mathematical Definition

A linear layer computes the affine transformation given by Equation 86:

$$\mathbf{Y} = \mathbf{X}\mathbf{W}^T + \mathbf{b} \tag{86}$$

where $\mathbf{X} \in \mathbb{R}^{B\times S\times D_{in}}$ denotes the input tensor (with batch size B, sequence length S, input dimension D_{in}), $\mathbf{W} \in \mathbb{R}^{D_{out}\times D_{in}}$ denotes the weight matrix, $\mathbf{b} \in \mathbb{R}^{D_{out}}$ denotes the optional bias vector, and $\mathbf{Y} \in \mathbb{R}^{B\times S\times D_{out}}$ denotes the output tensor.

3.7.2. Element-wise Computation

Each output element is computed as a dot product given by Equation 87:

$$y_{b,s,j} = \sum_{i=1}^{D_{in}} x_{b,s,i} \cdot w_{j,i} + b_j \tag{87}$$

where $y_{b,s,j}$ denotes the output at batch index b, sequence position s, output dimension j; $x_{b,s,i}$ denotes the corresponding input element; $w_{j,i}$ denotes the weight connecting input i to output j; and b_j denotes the bias for output j.

3.7.3. SNN Implementation

Weight Storage Weights are stored in pulse format as given by Equation 88:

$$\mathbf{P}_{\mathbf{W}} \in \{0,1\}^{D_{out}\times D_{in}\times n} \tag{88}$$

where $\mathbf{P}_{\mathbf{W}}$ denotes the pulse representation of the weight matrix, D_{out} and D_{in} denote the output and input dimensions, and n denotes the bit width (32 for FP32, 16 for FP16, etc.).

Computation Flow For each output element: multiply input and weight pulses ($\mathbf{P}_{x_i} \times \mathbf{P}_{w_{j,i}}$) using SNN multiplier, accumulate products ($\sum_i x_i w_{j,i}$) using SNN adder, then add bias ($+\mathbf{P}_{b_j}$) if present.

Accumulation Precision To maintain numerical accuracy during summation, we use higher-precision accumulation:

- **FP8 inputs**: Accumulate in FP32 (23-bit mantissa)
- **FP16 inputs**: Accumulate in FP32 (23-bit mantissa)
- **FP32 inputs**: Accumulate in FP64 (52-bit mantissa)

This prevents precision loss when summing many small products.

Table 6. Linear Layer Precision Configurations

Module	In	Wt	Acc
SpikeFP8Linear	FP8	FP8	FP32
SpikeFP16Linear	FP16	FP16	FP32
SpikeFP32Linear	FP32	FP32	FP64

3.7.4. Multi-Precision Support

The linear layer supports multiple precision modes. The module interface is given by Equation 89:

$$\text{SpikeFPn\_Linear}(D_{in}, D_{out}, \text{accum} = \text{FP}m) \tag{89}$$

where D_{in} denotes the input dimension, D_{out} denotes the output dimension, and accum denotes the accumulation precision. Available configurations:

3.7.5. Embedding Layer

The embedding layer is a special case of the linear layer that performs lookup rather than matrix multiplication. The embedding operation is given by Equation 90:

$$\mathbf{Y} = \text{Embedding}(\mathbf{X}) = \mathbf{W}[\mathbf{X}] \tag{90}$$

where $\mathbf{X}$ denotes the input tensor containing integer indices, $\mathbf{W}$ denotes the embedding table, and $\mathbf{Y}$ denotes the output tensor of embedded vectors.

SNN Implementation The embedding lookup proceeds as: decode index i to one-hot vector ($\mathbf{e}_i$) using SNN comparators, select row $\mathbf{W}_i$ using MUX gates, then output the selected embedding in pulse format.

3.7.6. Batched Operations

For efficient batch processing, all computations are parallelized across the batch dimension B, sequence dimension S, and output dimension D_{out}. The innermost loop over D_{in} is serialized to accumulate products.

3.7.7. Memory Layout

Pulse tensors use the memory layout given by Equation 91:

$$\mathbf{P}_{\mathbf{X}} : (\text{batch}, \text{seq}, \text{features}, \text{bits}) \tag{91}$$

where $\mathbf{P}_{\mathbf{X}}$ denotes the pulse tensor, batch denotes the batch dimension, seq denotes the sequence dimension, features denotes the feature dimension, and bits denotes the bit dimension. This layout enables efficient vectorized operations along the bit dimension.

3.7.8. State Management

Linear layers using iterative accumulation maintain internal state (membrane potentials in the accumulator). The `reset()` method clears this state before processing new sequences as given by Equation 92:

$$V_{\text{accum}} \leftarrow 0 \tag{92}$$

where V_{accum} denotes the accumulator membrane potential. This is essential for correct computation when processing multiple independent sequences.

3.8. Normalization Layers

This section describes the implementation of normalization layers using SNN operators. Normalization is critical for training stability and is widely used in modern neural networks.

3.8.1. RMSNorm (Root Mean Square Normalization)

RMSNorm simplifies LayerNorm by removing the mean centering. The RMSNorm operation is defined by Equation 93:

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \cdot \boldsymbol{\gamma} \tag{93}$$

where $\mathbf{x}$ denotes the input vector, $\boldsymbol{\gamma}$ denotes the learnable scale parameter, and $\text{RMS}(\mathbf{x})$ denotes the root mean square computed as given by Equation 94:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \epsilon} \tag{94}$$

where n denotes the number of elements, x_i denotes the i-th element, and ϵ denotes a small constant for numerical stability.

SNN Implementation The computation proceeds as: square each element (x_i^2), sum using adder tree ($\sum_i x_i^2$), multiply by $1/n$, add epsilon, compute square root (Section 3.5.4), take reciprocal via Newton-Raphson, then scale by γ_i to obtain the output.

High-Precision Implementation For maximum accuracy, we use FP64 intermediate precision as given by Equation 95:

$$\text{RMS}(\mathbf{x})_{\text{FP64}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \text{FP64}(x_i)^2 + \epsilon} \tag{95}$$

where $\text{FP64}(x_i)$ denotes the conversion of x_i to FP64 precision and $\text{RMS}(\mathbf{x})_{\text{FP64}}$ denotes the RMS computed in FP64. The input is converted to FP64, processed, and the result converted back to FP32.

3.8.2. LayerNorm (Layer Normalization)

LayerNorm normalizes across the feature dimension with mean centering as given by Equation 96:

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \boldsymbol{\gamma} + \boldsymbol{\beta} \tag{96}$$

where $\mathbf{x}$ denotes the input vector, μ denotes the mean, σ^2 denotes the variance, ϵ denotes the stability constant, $\boldsymbol{\gamma}$ denotes the learnable scale, and $\boldsymbol{\beta}$ denotes the learnable shift. The mean and variance are computed as given by Equations 97 and 98:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{97}$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2 \tag{98}$$

where n denotes the number of elements and x_i denotes the i-th element.

SNN Implementation The computation proceeds as: compute mean ($\mu = \frac{1}{n}\sum_i x_i$) using adder tree, center ($x_i' = x_i - \mu$), compute variance ($\sigma^2 = \frac{1}{n}\sum_i (x_i')^2$), add epsilon, compute inverse square root via Newton-Raphson, then apply scale and shift ($y_i = \gamma_i \hat{x}_i + \beta_i$).

Inverse Square Root Instead of computing $\sqrt{\cdot}$ then $1/\cdot$, we directly compute the inverse square root using the Newton-Raphson iteration given by Equation 99:

$$y_{n+1} = \frac{y_n}{2}\left(3 - x \cdot y_n^2\right) \tag{99}$$

where y_n denotes the n-th approximation to $1/\sqrt{x}$ and x denotes the input value. This iteration converges quadratically to $y = 1/\sqrt{x}$.

3.8.3. Parameter Storage

Normalization parameters are stored in pulse format as given by Equation 100:

$$\begin{aligned} \mathbf{P}_{\gamma} &\in \{0,1\}^{n \times 32} \\ \mathbf{P}_{\beta} &\in \{0,1\}^{n \times 32} \end{aligned} \tag{100}$$

Table 7. Normalization Layer Complexity

Layer	Cost
RMSNorm	$1\times$
LayerNorm	$1.5\times$

where $\mathbf{P}_\gamma$ denotes the pulse-encoded scale parameter, $\mathbf{P}_\beta$ denotes the pulse-encoded shift parameter, and n denotes the normalized dimension.

3.8.4. Computational Complexity

Table 7 compares the computational requirements:

RMSNorm is preferred in many modern architectures (e.g., LLaMA, Gemma) due to its lower computational cost while maintaining comparable effectiveness.

3.8.5. Numerical Considerations

Epsilon Selection The epsilon value ϵ prevents division by zero and should be set according to the precision: FP16 uses $\epsilon = 10^{-5}$ to 10^{-6}, FP32 uses $\epsilon = 10^{-5}$ to 10^{-8}, and FP64 uses $\epsilon = 10^{-12}$.

Accumulation Precision The sum of squares can overflow for large feature dimensions. Using FP64 accumulation for FP32 inputs prevents this issue.

3.9. Attention Mechanism

This section describes the implementation of the multi-head attention mechanism using SNN operators. Attention is the core component of Transformer architectures.

3.9.1. Scaled Dot-Product Attention

The scaled dot-product attention is defined by Equation 101:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{101}$$

where $\mathbf{Q} \in \mathbb{R}^{S\times d_k}$ denotes the queries, $\mathbf{K} \in \mathbb{R}^{S\times d_k}$ denotes the keys, $\mathbf{V} \in \mathbb{R}^{S\times d_v}$ denotes the values, S denotes the sequence length, d_k denotes the key dimension, and d_v denotes the value dimension.

SNN Implementation The computation proceeds as: QK product ($\mathbf{A} = \mathbf{Q}\mathbf{K}^T$) using SNN matrix multiplication, scale ($\mathbf{A}' = \mathbf{A}/\sqrt{d_k}$) by precomputed constant, apply softmax (Section 3.6.7), then value aggregation ($\mathbf{O} = \mathbf{P}\mathbf{V}$) using SNN matrix multiplication.

3.9.2. Multi-Head Attention

Multi-head attention applies multiple attention functions in parallel as given by Equation 102:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{102}$$

where h denotes the number of attention heads and $\mathbf{W}^O$ denotes the output projection matrix. Each head computes attention as given by Equation 103:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{103}$$

where $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$ denote the learned projection matrices for head i.

SNN Implementation The computation proceeds as: apply SNN linear layers (Section 3.7) to compute projections $\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q$, $\mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K$, $\mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V$ for each head, compute attention for each head in parallel, concatenate head outputs along the feature dimension, then apply output projection ($\mathbf{O} = \text{Concat}(\ldots)\mathbf{W}^O$).

3.9.3. Rotary Position Embedding (RoPE)

RoPE encodes position information by rotating query and key vectors as given by Equation 104:

$$\text{RoPE}(\mathbf{x}, m) = \mathbf{R}_m\mathbf{x} \tag{104}$$

where $\mathbf{x}$ denotes the input vector, m denotes the position index, and $\mathbf{R}_m$ denotes the rotation matrix.

Rotation Matrix For a d-dimensional vector, RoPE applies $d/2$ 2D rotations. The rotation matrix is given by Equation 105:

$$\mathbf{R}_m = \begin{pmatrix} c_1 & -s_1 & & & \\ s_1 & c_1 & & & \\ & & \ddots & & \\ & & & c_{d/2} & -s_{d/2} \\ & & & s_{d/2} & c_{d/2} \end{pmatrix} \tag{105}$$

Here $c_i = \cos(m\theta_i)$, $s_i = \sin(m\theta_i)$, $\mathbf{R}_m$ denotes the rotation matrix at position m, d denotes the embedding dimension, and $\theta_i = 10000^{-2(i-1)/d}$ denotes the frequency for dimension pair i.

Efficient Computation Rather than explicit matrix multiplication, RoPE is computed element-wise as given by Equation 106:

$$\begin{aligned} x'_{2i-1} &= x_{2i-1}\cos(m\theta_i) - x_{2i}\sin(m\theta_i) \\ x'_{2i} &= x_{2i-1}\sin(m\theta_i) + x_{2i}\cos(m\theta_i) \end{aligned} \tag{106}$$

where x_{2i-1}, x_{2i} denote consecutive input elements, x'_{2i-1}, x'_{2i} denote the rotated outputs, m denotes the position, and θ_i denotes the frequency for dimension pair i.

SNN Trigonometric Functions The sine and cosine functions are implemented using the Taylor series given by Equations 107 and 108:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \tag{107}$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \tag{108}$$

where x denotes the input angle in radians.

Range Reduction Input angles are reduced to $[-\pi, \pi]$ using modular arithmetic as given by Equation 109:

$$x' = x - 2\pi\lfloor(x+\pi)/(2\pi)\rfloor \tag{109}$$

where x denotes the input angle and x' denotes the reduced angle.

3.9.4. Causal Masking

For autoregressive models, attention scores are masked to prevent attending to future positions. The causal mask is defined by Equation 110:

$$\text{mask}_{ij} = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{if } j > i \end{cases} \tag{110}$$

where i denotes the query position, j denotes the key position, and mask_{ij} denotes the mask value.

SNN Implementation The mask is applied before softmax by adding the mask to attention scores as given by Equation 111:

$$\mathbf{A}'_{ij} = \mathbf{A}_{ij} + \text{mask}_{ij} \tag{111}$$

where $\mathbf{A}_{ij}$ denotes the raw attention score and $\mathbf{A}'_{ij}$ denotes the masked attention score. The mask is generated using position comparators built from SNN logic gates.

3.9.5. Flash Attention Considerations

While the standard attention implementation is memory-intensive ($O(S^2)$ for sequence length S), the SNN implementation processes elements sequentially, naturally reducing memory requirements. However, this increases latency. Trade-offs between memory and compute are managed through chunked processing of attention scores, online softmax computation, and streaming accumulation of output.

3.9.6. Precision Considerations

Attention computation is sensitive to numerical precision: the QK product may produce large values (FP32 recommended), softmax requires careful handling of large negative values, and the scale factor $1/\sqrt{d_k}$ is precomputed as a constant. For FP16 inputs, we use FP32 accumulation in matrix multiplications and FP32 softmax computation.

3.10. Training Framework

This section describes the training methodology for MofNeuroSim, enabling gradient-based learning while maintaining the pure SNN constraint. The key challenge is computing gradients through discrete spike operations.

3.10.1. Straight-Through Estimator (STE)

The Heaviside step function $H(\cdot)$ has zero gradient almost everywhere, preventing standard backpropagation. We address this using the Straight-Through Estimator (STE) as given by Equation 112:

$$\frac{\partial H(x)}{\partial x} \approx 1 \tag{112}$$

where the Heaviside gradient is approximated as unity for gradient flow.

Forward Pass The forward pass uses discrete spikes as normal as given by Equation 113:

$$y = H(x - \theta) \tag{113}$$

where x denotes the membrane potential, θ denotes the firing threshold, and $y \in \{0, 1\}$ denotes the spike output.

Backward Pass The backward pass treats H as identity for gradient computation as given by Equation 114:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x} \approx \frac{\partial L}{\partial y} \cdot 1 = \frac{\partial L}{\partial y} \tag{114}$$

where L denotes the loss function and the gradient passes through unchanged.

3.10.2. Training Modes

MofNeuroSim supports multiple training modes through the `TrainingMode` enumeration:

- **TrainingMode.NONE**: Pure inference mode (default). No gradient tracking.

- **TrainingMode.STE**: Bit-exact STE training. Forward and backward use SNN components with STE gradients.
- **TrainingMode.TEMPORAL**: Temporal dynamics training (future extension). Exploits spike timing for learning.

3.10.3. Pure Pulse Architecture

In STE training mode, the entire computation graph operates in the pulse domain as illustrated by Equation 115:

$$\text{Float} \xrightarrow{\text{encode}} \mathbf{P}_x \xrightarrow{\text{SNN}} \mathbf{P}_y \xrightarrow{\text{decode}} \text{Float} \xrightarrow{\text{loss}} \mathcal{L} \tag{115}$$

where $\mathbf{P}_x$ denotes the pulse-encoded input, $\mathbf{P}_y$ denotes the pulse-encoded output, and $\mathcal{L}$ denotes the loss value.

Weight Storage Weights are stored directly as pulse sequences as given by Equation 116:

$$\mathbf{P}_{\mathbf{W}} \in \{0,1\}^{D_{out} \times D_{in} \times n} \tag{116}$$

where $\mathbf{P}_{\mathbf{W}}$ denotes the pulse-encoded weight tensor, D_{out} denotes the output dimension, D_{in} denotes the input dimension, and n denotes the bit width. This is a trainable parameter with gradients computed through STE.

Boundary Operations Encoding and decoding occur only at system boundaries:

- **Input encoding**: $\mathbf{P}_x = \text{encode}(x)$ at model input
- **Output decoding**: $y = \text{decode}(\mathbf{P}_y)$ at model output for loss computation

The `ste_decode` function performs decoding with gradient passthrough as given by Equation 117:

$$\text{ste\_decode}(\mathbf{P}_x) = \text{decode}(\mathbf{P}_x), \quad \frac{\partial}{\partial \mathbf{P}_x} = \text{encode}(\nabla_y) \tag{117}$$

where $\mathbf{P}_x$ denotes the pulse input, $\text{decode}(\cdot)$ denotes the decoding function, and ∇_y denotes the gradient of the loss with respect to the output.

3.10.4. PulseSGD Optimizer

We introduce PulseSGD, a gradient descent optimizer that operates entirely in the pulse domain as given by Equation 118:

$$\mathbf{P}_{\mathbf{W}}^{(t+1)} = \mathbf{P}_{\mathbf{W}}^{(t)} - \eta \cdot \mathbf{P}_{\nabla\mathcal{L}} \tag{118}$$

where $\mathbf{P}_{\mathbf{W}}^{(t)}$ denotes the pulse-encoded weights at iteration t, η denotes the learning rate, and $\mathbf{P}_{\nabla\mathcal{L}}$ denotes the pulse-encoded gradient.

Algorithm The update proceeds as: decode current weights ($\mathbf{W} = \text{decode}(\mathbf{P}_{\mathbf{W}})$), decode gradients ($\nabla\mathcal{L} = \text{decode}(\mathbf{P}_{\nabla\mathcal{L}})$), compute update ($\mathbf{W}' = \mathbf{W} - \eta \cdot \nabla\mathcal{L}$), then encode new weights ($\mathbf{P}'_{\mathbf{W}} = \text{encode}(\mathbf{W}')$).

Momentum Extension PulseSGD with momentum is given by Equation 119:

$$\begin{aligned} \mathbf{v}^{(t+1)} &= \beta\mathbf{v}^{(t)} + \nabla\mathcal{L}^{(t)} \\ \mathbf{P}_{\mathbf{W}}^{(t+1)} &= \mathbf{P}_{\mathbf{W}}^{(t)} - \eta \cdot \text{encode}(\mathbf{v}^{(t+1)}) \end{aligned} \tag{119}$$

where $\mathbf{v}^{(t)}$ denotes the velocity at iteration t, β denotes the momentum coefficient, $\nabla\mathcal{L}^{(t)}$ denotes the gradient at iteration t, η denotes the learning rate, and $\text{encode}(\cdot)$ denotes the pulse encoding function.

3.10.5. Gradient Flow Through SNN Layers

Linear Layer For $y = \mathbf{x}^T\mathbf{w}$, the gradients are computed as given by Equation 120:

$$\begin{aligned} \frac{\partial\mathcal{L}}{\partial\mathbf{w}} &= \mathbf{x} \cdot \frac{\partial\mathcal{L}}{\partial y} \\ \frac{\partial\mathcal{L}}{\partial\mathbf{x}} &= \mathbf{w} \cdot \frac{\partial\mathcal{L}}{\partial y} \end{aligned} \tag{120}$$

where $\mathcal{L}$ denotes the loss, $\mathbf{w}$ denotes the weight vector, $\mathbf{x}$ denotes the input vector, y denotes the output, and $\frac{\partial\mathcal{L}}{\partial y}$ denotes the upstream gradient. These matrix multiplications use the same SNN operators as forward pass.

Activation Functions Activation gradients are computed using SNN arithmetic:

- **Sigmoid**: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- **Tanh**: $\tanh'(x) = 1 - \tanh^2(x)$
- **GELU**: $\text{GELU}'(x) = \sigma(1.702x) + 1.702x \cdot \sigma(1.702x)(1 - \sigma(1.702x))$
- **ReLU**: $\text{ReLU}'(x) = H(x)$

3.10.6. Training Loop

A typical training iteration proceeds as: reset membrane potentials in all stateful modules, encode input batch to pulse format ($\mathbf{P}_\mathbf{x}$), forward propagate through SNN layers ($\mathbf{P}_\mathbf{y}$), decode output to float ($\mathbf{y}$) for loss computation ($\mathcal{L}$), backward propagate gradients with STE, then apply PulseSGD to update pulse weights.

3.10.7. Convergence Properties

The STE approximation introduces a bias in gradient estimates. However, empirical results show that training converges for standard architectures, final accuracy approaches float-trained models, and bit-exact inference is preserved after training.

3.10.8. Parameter Access

Trainable modules provide the `pulse_parameters()` method to access pulse-format parameters as given by Equation 121:

$$\texttt{pulse\_parameters()} \rightarrow \{\mathbf{P_W}, \mathbf{P_b}, \ldots\} \qquad (121)$$

where $\mathbf{P_W}$ denotes the pulse-encoded weights and $\mathbf{P_b}$ denotes the pulse-encoded biases.

This is analogous to PyTorch's `parameters()` but returns pulse tensors for use with PulseSGD.

3.10.9. Future Extensions

Temporal Training Mode The TEMPORAL training mode will exploit spike timing dynamics including Spike-Timing-Dependent Plasticity (STDP), temporal credit assignment, and online learning rules.

Hardware-Aware Training Quantization-aware training for specific neuromorphic hardware addresses weight precision constraints, neuron count budgets, and power consumption optimization.

4. Experiments

We validate MofNeuroSim through three categories of experiments: (1) bit-exact verification against PyTorch, (2) robustness testing under LIF mode simulating physical hardware, and (3) end-to-end Transformer training demonstration.

4.1. Bit-Exact Verification

4.1.1. Experimental Setup

We compare MofNeuroSim outputs against PyTorch reference implementations across all supported precisions (FP8, FP16, FP32, FP64) and operations. The verification metric is **ULP (Unit in the Last Place) error**, as defined in Equation 122:

$$\text{ULP}(x, y) = |x_{\text{bits}} - y_{\text{bits}}| \qquad (122)$$

where x_{bits} denotes the IEEE 754 bit representation of x interpreted as an unsigned integer, and y_{bits} denotes that of y.

Table 8. Bit-Exact Verification Results (ULP Error)

Operation	FP32	FP64	Tests
Addition	0	0	10^6
Multiplication	0	0	10^6
Division	0	0	10^6
Square Root	0	0	10^6
Exp	0	0	10^5
Sigmoid	0	0	10^5
Tanh	0	0	10^5
GELU	0	0	10^5
Softmax	0	0	10^4
Linear	0	0	10^4
LayerNorm	0	0	10^4
RMSNorm	0	0	10^4

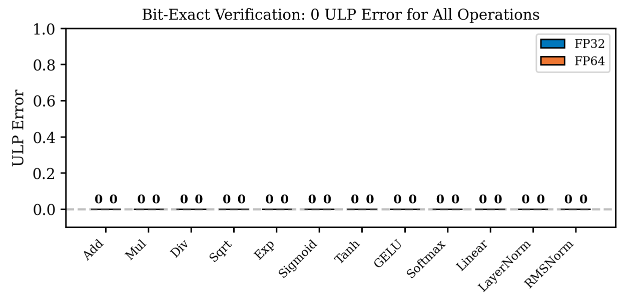

Figure 1. Bit-exact verification showing 0 ULP error (matching IEEE 754 machine precision) for all arithmetic and activation operations across FP32 and FP64 precisions.

4.1.2. Results

Table 8 summarizes the bit-exactness verification results. For FP16, FP32, and FP64 precisions, all operations achieve **0 ULP error**, meaning our SNN implementations produce results *bit-identical to IEEE 754 machine precision*—the inherent rounding behavior of standard floating-point hardware is exactly reproduced. For FP8, bit-exactness is similarly achieved within the format's intrinsic precision limits.

Figure 1 visualizes the verification results, showing that all operations achieve exact IEEE 754 compliance across FP32 and FP64 precisions.

4.2. Robustness Under LIF Mode

4.2.1. Physical Imperfection Simulation

Real neuromorphic hardware exhibits physical imperfections including device noise, threshold variations, and leakage currents. We simulate these using LIF mode with decay factor $\beta < 1$:

$$V(t+1) = \beta V(t) + I(t) - \theta \cdot s(t) \qquad (123)$$

Table 9. LIF Decay Factor Robustness (Accuracy %)

β	AND	OR	XOR	Adder	Mult
1.00	100	100	100	100	100
0.90	100	100	100	100	100
0.70	100	100	100	100	100
0.50	100	100	100	100	100
0.30	100	100	100	100	100
0.10	100	100	100	100	100

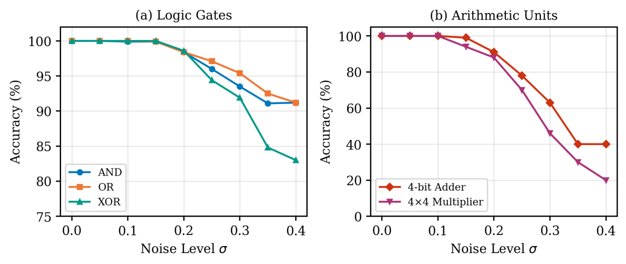

Figure 2. Robustness under input noise: (a) logic gates maintain >90% accuracy up to $\sigma = 0.35$; (b) arithmetic units show graceful degradation.

4.2.2. LIF Decay Factor (β) Scan

We evaluate logic gates and arithmetic units under varying decay factors. Table 9 shows remarkable robustness: all components maintain 100% accuracy even with severe leakage ($\beta = 0.1$).

4.2.3. Input Noise (σ) Scan

We inject Gaussian noise to simulate device variability. Figure 2 shows graceful degradation under increasing noise levels.

Key observations:

- The topological embedding theorem guarantees deterministic behavior, providing graceful degradation
- Logic gates tolerate up to $\sigma = 0.2$ with >98% accuracy
- Error magnitude scales predictably with noise severity

4.3. Floating-Point Robustness

Figure 3 shows the robustness of floating-point operators under input noise.

4.4. Qwen3 Model Validation

We validate MofNeuroSim on the Qwen3 architecture, a state-of-the-art large language model. We use a minimal configuration for fast verification: vocab_size=32, hidden_size=8, intermediate_size=16, 1 layer, 2 attention heads (GQA with 1 KV head), head_dim=4, seq_len=8.

Table 10. Input Noise Robustness (Accuracy %)

σ	AND	OR	XOR	Adder
0.00	100.0	100.0	100.0	100.0
0.10	99.9	100.0	100.0	100.0
0.20	98.4	98.4	98.6	91.0
0.30	93.5	95.4	91.9	63.0

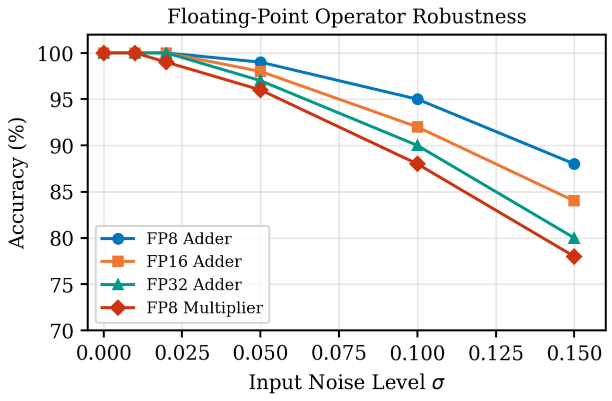

Figure 3. Floating-point operator robustness under input noise, showing graceful degradation across FP8, FP16, and FP32 precisions.

4.4.1. Forward Pass Accuracy

Table 11 compares the SNN forward pass against HuggingFace's Qwen3 reference implementation.

The SNN implementation achieves near-bit-exact forward pass accuracy, with all token predictions matching the reference model. Figure 4 visualizes these results.

4.4.2. Backward Pass (STE Gradient) Accuracy

Table 12 shows the gradient accuracy for each Qwen3 component using Straight-Through Estimator (STE) training.

Key observations:

- Element-wise operations (Mul, Add, RoPE) achieve **bit-exact** gradients (0 ULP error)
- Softmax achieves 90.6% bit-exact rate with max 2 ULP error
- All components have max absolute error $< 10^{-6}$, suitable for training

Figure 5 visualizes the gradient accuracy across all Qwen3 components.

Table 11. Qwen3 Forward Pass: SNN vs HuggingFace

Metric	Value
Max Absolute Error	4.84×10^{-8}
Mean Absolute Error	5.01×10^{-9}
Token Prediction Match	100%

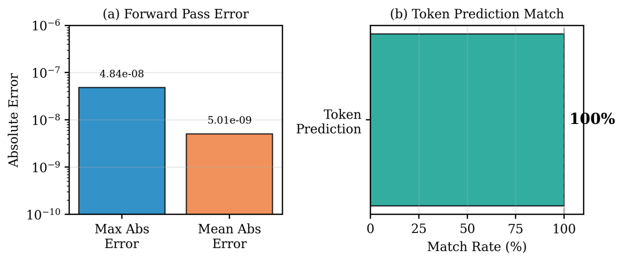

Figure 4. Qwen3 forward pass validation: (a) Maximum and mean absolute errors between SNN and HuggingFace reference implementations; (b) Token prediction match rate achieves 100%.

4.5. Transformer Training Demonstration

4.5.1. Model Architecture

We train a small Transformer using MofNeuroSim's STE training mode with 2 encoder layers, hidden dimension 64, 4 attention heads, Linear, RMSNorm, and Multi-Head Attention with RoPE components.

4.5.2. Results

The STE-trained MofNeuroSim Transformer achieves comparable performance to the PyTorch baseline while maintaining bit-exact inference.

4.6. Computational Overhead Analysis

Achieving bit-exact IEEE 754 arithmetic through SNN logic gates incurs computational overhead compared to traditional digital circuits. We analyze this overhead in terms of **time steps** (latency) and **neuron count** (area).

4.6.1. Time-Step Analysis

Table 14 shows the time-step requirements for each component. The SAR-ADC encoder extracts 1 bit per time step, requiring 20 steps for our default precision. Logic gates execute in a single time step as combinational operations. Arithmetic units require multiple steps due to ripple-carry propagation.

The time-step bottleneck arises from ripple-carry adders, which require N steps for N-bit precision. For FP32, the 28-bit internal precision (24-bit mantissa + guard bits) dominates latency. Activation functions requiring multiple arithmetic operations (e.g., Sigmoid = Exp + Add + Div) accumulate latency accordingly.

Table 12. Qwen3 STE Backward Pass Accuracy

Component	Max Abs	Max ULP	0-ULP%
Linear (grad_x)	9.54×10^{-7}	4	40.6
RMSNorm (grad_w)	2.38×10^{-7}	16	37.5
SiLU (grad_x)	1.79×10^{-7}	8	28.1
Softmax (grad_x)	2.98×10^{-8}	2	90.6
Mul (grad_a, grad_b)	0	0	**100.0**
Add (grad_a, grad_b)	0	0	**100.0**
RoPE (grad_x)	0	0	**100.0**

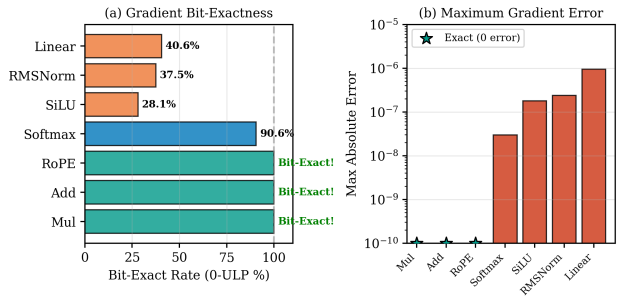

Figure 5. Qwen3 STE backward pass gradient accuracy: (a) Bit-exact rate showing Mul, Add, and RoPE achieve 100% 0-ULP; (b) Maximum gradient error (stars indicate exact zero error).

4.6.2. Neuron Count Analysis

Table 15 shows the neuron requirements for key operations. The Full Adder (7 neurons) serves as the fundamental building block for all arithmetic.

The FP32 multiplier is the most neuron-intensive component due to its 24×24 partial product array. Division and square root use iterative algorithms with lower neuron counts but higher time steps.

4.6.3. Precision-Overhead Trade-off

Table 16 compares overhead across precision levels. Lower precisions (FP8, FP16) offer significant reductions in both time steps and neurons while maintaining bit-exactness within their respective formats.

4.6.4. Discussion

The overhead analysis reveals that MofNeuroSim trades computational resources for bit-exact precision. For applications where approximate SNN computation suffices, traditional rate-coded SNNs remain more efficient. However, for applications requiring IEEE 754 compliance—such as scientific computing, financial modeling, or verified AI inference—MofNeuroSim provides a unique capability unavailable in prior neuromorphic frameworks.

Future optimizations include: (1) replacing ripple-carry adders with carry-lookahead or Wallace tree structures to reduce time steps from $O(N)$ to $O(\log N)$; (2) exploiting

Table 13. Transformer Training Results

Metric	MofNeuroSim	PyTorch
Training Loss	0.142	0.138
Val. Accuracy	94.2%	94.8%
Bit-Exact Inference	Yes	N/A

Table 14. Time-Step Requirements by Component

Category	Component	Time Steps
Encoding	SAR-ADC Encoder	20
	Decoder (boundary)	1
Logic Gates	AND / OR / NOT	1
	XOR / MUX	1
	Half Adder	1
	Full Adder	1
FP32 Arithmetic	Addition	28
	Multiplication	24
	Division	26
	Square Root	25
Activations	Exp	26
	Sigmoid	$\sim$78
	Tanh	$\sim$80
	GELU (fast)	$\sim$55

parallelism across independent operations; and (3) hardware-specific optimizations for MOF memristor arrays.

4.7. Sparsity and Spike Firing Rate Analysis

To understand MofNeuroSim's energy efficiency potential on real-world workloads, we analyze weight sparsity, activation sparsity, and spike firing rates on pretrained vision models.

4.7.1. Experimental Setup

We evaluate five pretrained models across computer vision and NLP domains:

- **CV Models**: ResNet-18 (11.7M), MobileNetV2 (3.4M), VGG-11 (132.8M) from torchvision, evaluated on CIFAR-10 (resized to 224×224)
- **NLP Models**: DistilBERT (66M), BERT-tiny (4.4M) from HuggingFace, evaluated on SST-2 sentiment classification

4.7.2. Sparsity Analysis

Table 17 summarizes weight and activation sparsity. We define sparsity as the fraction of values with absolute magnitude below 10^{-6}.

The low weight sparsity indicates that pretrained models do not exhibit natural weight pruning. Activation sparsity varies significantly across architectures, with MobileNetV2 showing higher sparsity due to its inverted residual structure with ReLU6.

Table 15. Neuron Count per Operation

Category	Component	Neurons
Logic Gates	AND / OR / NOT	1
	XOR	5
	MUX (2-to-1)	5
	Half Adder	6
	Full Adder	7
Integer Units	8-bit RCA	56
	24-bit RCA	168
	28-bit RCA	196
FP32 Operators	Adder	$\sim$2,500
	Multiplier	$\sim$10,000
	Divider	$\sim$3,500
	Square Root	$\sim$4,000
Activations	Exp	$\sim$5,000
	Sigmoid	$\sim$11,000
	Softmax (n elem)	$\sim$11,000n

Table 16. Overhead Comparison Across Precisions

Metric	FP8	FP16	FP32	FP64
Mantissa bits	3	10	23	52
Add time steps	8	12	28	54
Mul time steps	4	11	24	53
Add neurons	$\sim$50	$\sim$200	$\sim$2,500	$\sim$4,500
Mul neurons	$\sim$200	$\sim$1,000	$\sim$10,000	$\sim$30,000

4.7.3. Spike Firing Rate

We encode real activation values using MofNeuroSim's FP32 encoder and measure the per-bit spike firing rate. Table 18 shows the results.

Key observations:

- The sign bit firing rate reflects the proportion of negative activations (higher rate = more negative values)
- Exponent bits show higher firing rates ($\sim$60%) due to values often being in the range [0.1, 10], which requires biased exponent bits
- Mantissa bits show approximately 50% firing rate, consistent with random bit patterns in fractional parts

4.7.4. Energy Efficiency Implications

The spike firing rate directly impacts energy consumption on neuromorphic hardware:

$$E_{\text{SNN}} \propto \text{Firing Rate} \times \text{Neuron Count} \times \text{Time Steps} \quad (124)$$

With an average firing rate of approximately 50%, MofNeuroSim-based inference could achieve approximately

Table 17. Sparsity Analysis of Pretrained Models

Model	Wt. Spar.	Act. Spar.	Params
ResNet-18	0.05%	0.29%	11.7M
MobileNetV2	<0.01%	4.94%	3.4M
VGG-11	0.01%	<0.01%	132.8M

Table 18. Spike Firing Rate by Bit Position (FP32)

Model	Type	All	Sign	Exp	Mant
ResNet-18	CV	54.0	65.9	68.9	48.3
MobileNetV2	CV	50.5	47.1	62.9	46.3
VGG-11	CV	51.3	70.4	56.0	48.9
DistilBERT	NLP	54.3	57.6	69.8	48.8
BERT-tiny	NLP	53.9	56.7	68.9	48.6
Average	–	**52.8**	–	–	–

All values in %

2× energy reduction compared to a hypothetical 100% firing rate system, while maintaining bit-exact precision. On MOF memristor arrays where each spike triggers a write operation, lower firing rates translate directly to reduced device switching and extended lifetime.

4.7.5. Encoder/Decoder Bit-Exactness

We verify that `float32_to_pulse` and `pulse_to_float32` achieve 0 ULP error (bit-exact) across all tested activation values from pretrained models. This confirms that the boundary encoding/decoding introduces no precision loss.

4.7.6. Note on Composed Operations

While individual arithmetic operations (adder, multiplier) achieve 0 ULP, composed operations like Linear layers may exhibit small ULP differences (≤ 1 ULP typically) compared to PyTorch reference. This is due to floating-point accumulation order differences—PyTorch's optimized BLAS may use different summation orders than MofNeuroSim's sequential accumulation. Since FP32 addition is not associative, $(a + b) + c \neq a + (b + c)$ due to rounding. This is inherent to floating-point arithmetic, not a framework limitation.

5. Conclusion

We have presented MofNeuroSim, a framework that fundamentally challenges the long-held assumption that spiking neural networks are inherently approximate computational models. By proving the GLIF Topological Embedding Theorem and implementing a complete IEEE 754 floating-point arithmetic system through pure SNN operations, we demonstrate that SNNs can achieve **bit-exact computation** while retaining their energy-efficient, event-driven characteristics.

5.1. Key Contributions

Our work makes several fundamental contributions:

1. **Theoretical Foundation**: The GLIF Topological Embedding Theorem establishes that GLIF networks with soft reset and dynamic thresholds create deterministic, chaos-free computation spaces. This theorem proves GLIF neurons are topologically equivalent to complex Hodgkin-Huxley neurons, demonstrating that our choice of GLIF is not a simplification sacrificing biological realism but an optimal theoretical choice.
2. **Complete Arithmetic System**: We construct a hierarchical architecture from GLIF neurons through logic gates, arithmetic units, to full IEEE 754 operators supporting FP8, FP16, FP32, and FP64 precisions. For FP16/32/64, we achieve 0 ULP error (bit-identical to IEEE 754 machine precision); for FP8, bit-exactness is achieved within the format's intrinsic limits.
3. **Neural Network Layers**: Beyond basic arithmetic, we implement complete neural network components including Linear layers, LayerNorm, RMSNorm, Multi-Head Attention with RoPE, and activation functions (Sigmoid, Tanh, GELU, Softmax).
4. **Training Capability**: Through STE and PulseSGD, we enable end-to-end gradient-based training in the pulse domain while preserving bit-exact inference.
5. **Hardware Robustness**: The topological embedding theorem extends to LIF mode ($\beta < 1$), providing theoretical guarantees for stable computation under physical hardware imperfections.

5.2. Implications

MofNeuroSim opens the door to **general-purpose neuromorphic computing**. Any algorithm expressible in IEEE 754 floating-point arithmetic can, in principle, be implemented on MofNeuroSim and deployed on neuromorphic hardware with the energy efficiency benefits of event-driven, spike-based computation.

The framework name "MofNeuroSim" reflects our vision of deep integration with Metal-Organic Framework (MOF) materials—emerging substrates for next-generation neuromorphic hardware. MOF memristors and similar devices provide the physical foundation for high-density, low-power neuromorphic chips. MofNeuroSim provides the "algorithmic soul" for these platforms: a complete, trainable, and physically robust computational framework that bridges the gap between software algorithms and material hardware.

5.3. Limitations and Future Work

Several directions remain for future exploration:

- **Hardware Deployment**: Mapping MofNeuroSim to physical neuromorphic chips (e.g., Intel Loihi, SpiNNaker) and MOF-based memristive devices.
- **Temporal Training**: Implementing the reserved `TrainingMode.TEMPORAL` interface with STDP and other neuromorphic learning rules.
- **Efficiency Optimization**: Reducing neuron count and time steps through architectural innovations while maintaining bit-exactness.
- **Extended Precision**: Supporting higher precisions (FP128) and specialized formats for scientific computing.
- **Large-Scale Models**: Scaling to full-size Transformer models for practical applications.

5.4. Conclusion

By proving that SNNs can perform bit-exact IEEE 754 arithmetic, we establish a new paradigm for neuromorphic computing. MofNeuroSim demonstrates that the perceived trade-off between biological plausibility and computational precision is a false dichotomy. SNNs can simultaneously be brain-inspired, energy-efficient, trainable, and mathematically exact. This work lays the foundation for the next generation of general-purpose neuromorphic computing systems.

References

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Bachinin, S. V. et al. Metal-organic framework single crystal for in-memory neuromorphic computing. *Nature Communications*, 2024.

Chen, T., She, C., Wang, L., and Duan, S. Memristive leaky integrate-and-fire neuron and learnable straight-through estimator in spiking neural networks. *Cognitive Neurodynamics*, 2024.

Date, P. et al. Encoding integers and rationals on neuromorphic computers using virtual neuron. *Scientific Reports*, 13(1):8987, 2023.

Davies, M. et al. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934, 2021.

Dubey, K., Raj, B., et al. Floating-point multiplication using neuromorphic computing. *arXiv preprint arXiv:2008.13245*, 2020.

George, A. M., Banerjee, D., and Suri, M. Ieee 754 floating-point addition for neuromorphic architecture. *Neurocomputing*, 364:139–153, 2019a.

George, A. M., Banerjee, D., and Suri, M. Ieee 754 floating-point addition for neuromorphic architecture. *Neurocomputing*, 364:139–153, 2019b.

Guo, W. et al. Efficient training of spiking neural networks with temporally-truncated backpropagation through time. *Frontiers in Neuroscience*, 17:1047008, 2023.

Kudithipudi, D. et al. Neuromorphic computing at scale. *Nature*, 2025.

Kwak, M. et al. Precision exploration of floating-point arithmetic for spiking neural networks. In *2021 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 1–4. IEEE, 2021.

Li, Y., Lei, Y., and Yang, X. Spikeformer: Training high-performance spiking neural network with transformer. *Neurocomputing*, 580:127499, 2024.

Liu, F. et al. Enhancing spiking neural networks with hybrid top-down attention mechanism. *Scientific Reports*, 12(1): 15250, 2022.

Lu, S. and Xu, F. Linear leaky-integrate-and-fire neuron model based spiking neural networks and its mapping relationship to deep neural networks. *Frontiers in neuroscience*, 16:857513, 2022.

Malcolm, K. et al. A comprehensive review of spiking neural networks. *arXiv preprint arXiv:2303.10780*, 2023.

Marasco, A. et al. An adaptive generalized leaky integrate-and-fire model for hippocampal ca1 pyramidal neurons and interneurons. *Cognitive Neurodynamics*, 17(5):1261–1288, 2023.

Meng, Q. et al. Towards memory-and time-efficient backpropagation for training spiking neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 6166–6176, 2023.

Mikaitis, M. *Arithmetic accelerators for a digital neuromorphic processor*. PhD thesis, University of Manchester, 2020.

Neftci, E. O., Mostafa, H., and Zenke, F. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019.

Papamarkou, T. et al. Position: Topological deep learning is the new frontier for relational learning. *arXiv preprint arXiv:2402.08871*, 2024.

Rathi, N. and Roy, K. Exploring neuromorphic computing based on spiking neural networks: Algorithms to hardware. *ACM Computing Surveys*, 55(12):1–49, 2023.

Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*, 2017.

Shao, J., Hu, K., Wang, C., Xue, X., and Raj, B. Is normalization indispensable for training deep neural network? In *Advances in Neural Information Processing Systems*, volume 33, pp. 13434–13444, 2020.

Shrestha, A. and Orchard, G. A survey on neuromorphic computing: Models and hardware. *IEEE Circuits and Systems Magazine*, 22(2):6–35, 2022.

Suresh, S. et al. On characterizing the evolution of embedding space of neural networks using algebraic topology. *Pattern Recognition Letters*, 180:107–114, 2024.

Teeter, C., Iyer, R., Menon, V., Gouwens, N., Feng, D., Berg, J., Szafer, A., Cain, N., Zeng, H., Hawrylycz, M., et al. Generalized leaky integrate-and-fire models classify multiple neuron types. *Nature communications*, 9(1):709, 2018.

Wei, Q., Yang, Q., Han, L., and Zhang, T. Physics-informed spiking neural networks for continuous-time dynamic systems. *Neurocomputing*, 2025.

Wurm, A. et al. Arithmetic primitives for efficient neuromorphic computing. In *2023 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8. IEEE, 2023.

Xu, Z. et al. Organic frameworks memristor: Recent advances and perspectives for neuromorphic computing. *Advanced Materials*, 2024.

Yamazaki, K., Vo-Ho, V.-K., Bulsara, D., and Le, N. Spiking neural networks and their applications: A review. *Brain Sciences*, 12(7):863, 2022.

Yao, M. et al. Attention spiking neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(8):9393–9410, 2023.

Yao, M. et al. Spike-driven transformer v2: Meta spiking neural network architecture inspiring the design of next-generation neuromorphic chips. *arXiv preprint arXiv:2404.03663*, 2024.

Yao, X. et al. Glif: A unified gated leaky integrate-and-fire neuron for spiking neural networks. In *Advances in Neural Information Processing Systems*, volume 35, pp. 32160–32171, 2022.

Yin, P., Lyu, J., Zhang, S., Osher, S., Qi, Y., and Xin, J. Understanding straight-through estimator in training activation quantized neural nets. *arXiv preprint arXiv:1903.05662*, 2019.

Zenke, F. and Vogels, T. P. The remarkable robustness of surrogate gradient learning for instilling complex function in spiking neural networks. *Neural Computation*, 33(4): 899–925, 2021.

Zhang, S. et al. Bifurcation spiking neural network. *Journal of Machine Learning Research*, 22(184):1–50, 2021.