Preprint of paper from ISVLSI 2021.

A Microarchitecture Implementation Framework for Online Learning with Temporal Neural Networks

Harideep Nair
Electrical and Computer Engineering
Carnegie Mellon University
harideep.nair@sv.cmu.edu

John Paul Shen
Electrical and Computer Engineering
Carnegie Mellon University
jpshen@cmu.edu

James E. Smith
Electrical and Computer Engineering
University of Wisconsin (Emeritus)
Carnegie Mellon University (Adjunct)
jes@ece.wisc.edu

***Abstract*—Temporal Neural Networks (TNNs) are spiking neural networks that use time as a resource to represent and process information, similar to the mammalian neocortex. In contrast to compute-intensive deep neural networks that employ separate training and inference phases, TNNs are capable of extremely efficient online incremental/continual learning and are excellent candidates for building edge-native sensory processing units. This work proposes a microarchitecture framework for implementing TNNs using standard CMOS. Gate-level implementations of three key building blocks are presented: 1) multi-synapse *neurons*, 2) multi-neuron *columns*, and 3) unsupervised and supervised online learning algorithms based on *Spike Timing Dependent Plasticity (STDP)*. The proposed microarchitecture is embodied in a set of characteristic scaling equations for assessing the gate count, area, delay and power for any TNN design. Post-synthesis results (in 45nm CMOS) for the proposed designs are presented, and their online incremental learning capability is demonstrated.**

***Index Terms*—temporal neural networks, online learning**

I. Introduction

Current computing demand for training Deep Neural Networks (DNNs) is doubling every 3.4 months [20]. Moore's law, at best, is only doubling every 2 years. The gap between increasing computing demand and what computing hardware can provide is widening at the rate of 8x per year. This calls for new paradigms and new types of hardware that are orders of magnitude more efficient for performing human-like sensory processing and online learning [22]. Neuromorphic temporal neural networks appear to exhibit such potential.

Temporal Neural Networks (TNNs) [24]–[26] strive to achieve not just the behavior/function of biological neural networks but also their structure/organization. TNNs adhere to biological plausibility with the goal of achieving brain-like capability and efficiency. Fig. 1 highlights the distinctive neuromorphic attributes of TNNs. TNN components communicate via spikes like all Spiking Neural Networks (SNNs) [15], [19]. However, TNNs belong to a special class of SNNs that encode and process information in temporal form using precise *spike time relationships*, unlike most SNNs that use *spike rates* [4], [10], [23] for information encoding and processing. TNNs also employ a form of *local* learning called Spike Timing Dependent Plasticity (STDP) [7], as opposed to *global* backpropagation and stochastic gradient descent commonly used in DNNs [13] and SNNs [2], [14], [17].

TNNs fueled by STDP are capable of learning in an online, incremental, continual fashion [25], [26] and therefore provide a promising technology for building sensory processing units in mobile/edge devices. The efficacy of TNNs in performing unsupervised time-series clustering for various edge-native applications such as anomaly detection, healthcare monitoring, etc. has been shown in [3]. This work builds on recent work in [24] which lays the foundation of TNNs as space-time computing networks based on a rigorous space-time algebra. In [25], [26] it is proposed that one can implement a *silicon neocortex* capable of brain-like online learning by examining the organization of biological neural networks to formulate an analogous architecture for TNNs. We follow this approach by focusing on direct hardware implementation of TNNs.

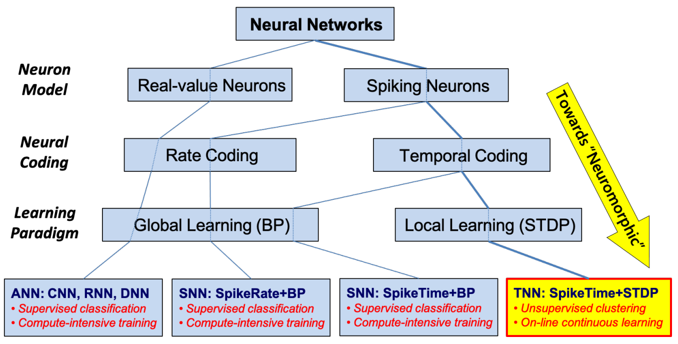


Figure 1: Neural Network Taxonomy

This work explores the practical feasibility of direct hardware implementation of TNNs using standard digital CMOS technology. In a direct implementation, hardware clock cycle is used as the basic time unit for temporal processing, i.e., time itself is not stored as a binary value but implicit in the clock. We define a TNN microarchitecture and implement its key building blocks: 1) multi-synapse *neurons*, 2) multi-neuron *columns* and 3) *STDP* (unsupervised) and *R-STDP* (supervised with reward) online learning algorithms. We present their gate-level designs along with characteristic scaling equations for estimating the area, power and delay for any arbitrary TNN. A distinct feature of the proposed framework is a novel synapse design that integrates weight storage with synaptic processing, thereby eliminating the need for a separate weight storage. To the best of our knowledge, this is the first work that presents a microarchitecture framework for directly implementing TNNs capable of online learning.

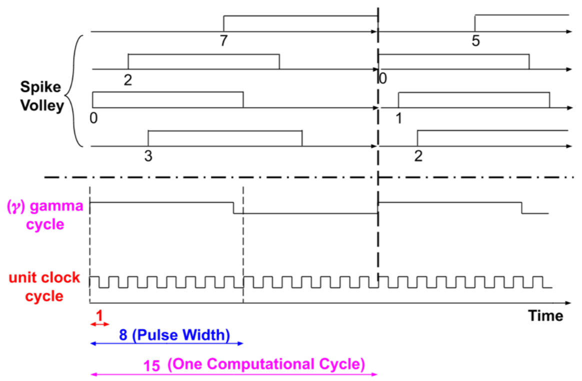
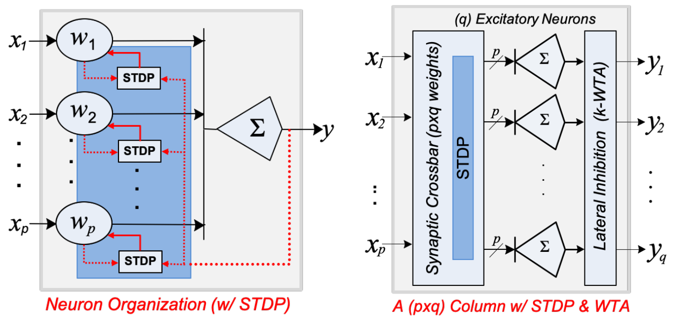


Figure 2: Temporal Encoding and Processing


x_1
w_1
STDP
x_2
w_2
x_p
w_p
Σ
y
Neuron Organization (w/ STDP)
(q) Excitatory Neurons
Synaptic Crossbar (pxq weights)
STDP
Lateral Inhibition (k-WTA)
y_1
y_2
y_q
A (pxq) Column w/ STDP & WTA

(a) Neuron: p Synapses, STDP (b) Column: q Neurons & WTA

Figure 3: Key TNN Building Blocks

II. TNN Organization and Operation

A. Temporal Encoding and Processing

A distinctive attribute of TNNs involves the use of temporal encoding, wherein information is represented by relative timings of spikes. In a TNN, computation occurs in volleys or waves of spikes. A volley consists of at most one spike per synaptic input. As shown in Fig. 2, two clocks are used. The *unit clock* is the finest temporal resolution and is also the synchronizing clock used in the digital hardware. The *gamma clock* (inspired from the biological gamma cycles [6]) frames the computing window and is the time required for a column to communicate and process a spike volley and update synaptic weights. This implementation studied here uses 3 bits of precision for temporal encoding and synaptic weights. Spikes in a volley are implemented as unit time pulses, a form of unary encoding, and volleys are separated using gamma clock cycles. With unary encoding, it takes up to 7 time units to encode a 3-bit value. Allowing additional time to process a spike volley, the gamma cycle is extended to 15 time units. This is explained in further detail in Section III-B.

B. Key TNN Building Blocks

The most fundamental TNN building block is a neuron. As shown in Fig. 3a, each neuron has p synaptic inputs and one output. Each synaptic input carries a synaptic weight, which is updated locally based on the relative timing of the incoming spike to that synapse and the outgoing spike from the associated neuron body. The rules for updating synaptic weights constitute the STDP learning method - the key building block that imparts TNNs their functionality. Through STDP, a neuron learns an input feature by adapting its synaptic weights to closely match the corresponding input pattern.

The smallest operational building block is a column which, in itself, is a fully-functional TNN. As shown in Fig. 3b, a column is a stack of q parallel neurons. Every neuron in a column shares the same set of p inputs, known as a *receptive field*. A $p\times q$ synaptic crossbar contains $p\times q$ synaptic weights, each of which is updated by STDP. On the output side of the q neurons, one winner-take-all (1-WTA) *lateral inhibition* is performed by selecting the earliest spiking neuron from among the q neurons as the one winner. Output spiking is disabled for non-winning neurons. This introduces competition among the neurons and enables the column to learn a set of distinct features local to its input receptive field.

This paper presents the CMOS implementation of a neuron (Section III) and a column (Section V). In Section IV, STDP rules for updating synaptic weights are discussed. The baseline STDP method is unsupervised. We also introduce a variation, called *reinforcement* STDP, which is similar to the *reward modulated* STDP in [18]. Post-synthesis and online learning evaluations are performed in Sections VI and VII respectively.

III. Neuron Implementation

This work adopts the widely used Spike Response Model [11] SRM0. This section presents the components of this excitatory neuron model and their detailed gate level designs.

A. Synaptic Response Functions

A *synapse* connects the *axon* (output) of a pre-synaptic neuron and a *dendrite* (input) of the post-synaptic neuron. An SRM0 neuron takes multiple input spikes and generates a response function for each spike based on its corresponding synaptic weight. All the individual response functions are then added to form the neuron's membrane potential. When (and if) the membrane potential crosses a threshold, the neuron fires an output spike on its axon. In this work, we adopt the ramp-no-leak (RNL) function for its temporal computational benefits and implementation efficiency [15], [25]. The RNL function increases by a unit step at every time unit until it reaches its peak and then remains constant until it is reset prior to the next computation cycle. The "ramp" allows responses from different synapses to be distributed temporally based on the synaptic strengths (weights), which proves to be particularly powerful for TNNs that operate temporally. The no-leak model is based on arguments that the leak is primarily a reset mechanism [7], [16]. For silicon implementations, there are simpler ways to reset at the end of each gamma cycle.

B. Synapse Modeling

Fig. 4 shows the block diagram for the proposed SRM0 neuron implementing RNL response function. Its operation consists of three main stages: 1) temporal arrival of input

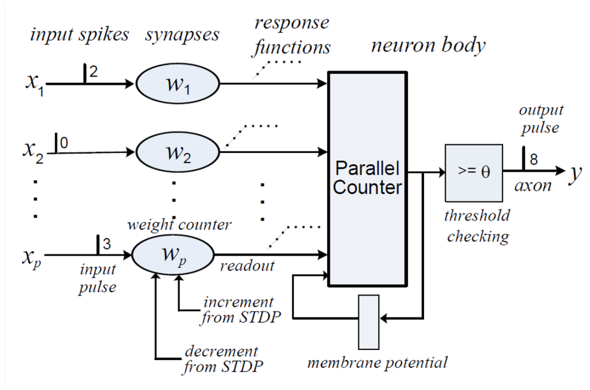


Figure 4: SRM0 Neuron with RNL Response Function

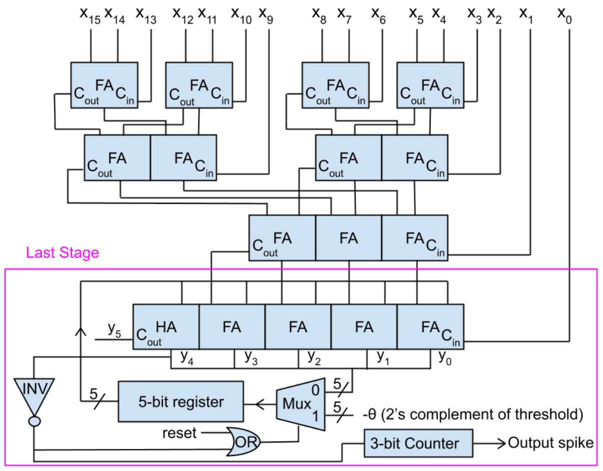


Figure 5: Neuron Body with 16 Synapses

spikes, 2) serial thermometer readout of RNL response functions based on the corresponding synaptic weights, and 3) binary accumulation of thermometer-coded response functions into the membrane potential. Synapses are implemented as finite state machines (FSMs) operating as binary counters. If the maximum weight is w_{max}, the number of counter bits is $ceiling(log_2(w_{max}+1))$. The counter has three modes, two controlled by STDP (described in Section IV): increment (up to w_{max}) and decrement (down to 0). The third *readout* mode is controlled by the input pulse. Readout mechanism is meticulously integrated into the same FSM used for storing synaptic weight and is described below.

As will become apparent, synapses dominate hardware complexity and hence the synapse design must be highly optimized. Our approach uses a pulse of width $w_{max}+1$. The input pulse directly controls the counter readout. When the leading edge of an input pulse occurs (0→1 transition), the weight counter is decremented and an output of 1 is emitted each unit clock cycle until the counter reaches 0. This essentially converts the binary weight value in the counter to a serial thermometer code. After the counter reaches 0, it wraps around to w_{max} and continues to count down until the trailing edge of the input pulse (1→0 transition) when the weight in the counter is restored to its original value. Thus, once an input spike arrives, readout takes an additional 7 cycles. (Although we assume w_{max} = 7 in this paper, this technique can be generalized to any w_{max}.) STDP (Section IV) takes another cycle. These coupled with 7 cycles for encoding give rise to a gamma period of 15 clock cycles.

In summary, a synapse and its weight are implemented with a counter FSM that can 1) increment, saturating at w_{max}; 2) decrement, saturating at 0; and 3) wrap-around decrementing, emitting an output of 1 prior to wrapping around and then a 0 thereafter. Note that this synapse design preserves the original weight value while doing RNL readout, which eliminates the need of a separate SRAM for weight storage and access.

C. Neuron Body

The neuron body is implemented as a parallel counter that adds the thermometer coded weights coming from the synapses, cycle by cycle, thereby accumulating the membrane potential as a sum of RNL response functions. When (and if) the parallel counter output reaches the threshold θ, an output spike is emitted during that cycle.

Based on Parhami [21], the membrane potential accumulator can be efficiently implemented using ripple carry adders by integrating a (p-1)-input parallel combinational counter and a (log_2p+1)-bit adder into one design. Fig. 5 shows the design for a 16-input accumulator, with integrated output spike generation. For a p-input accumulator, p-1 inputs are accumulated into a (log_2p)-bit output, which is then added to the previous stored (log_2p+1)-bit value from the register with the one remaining input bit acting as carry-in. Note that the configuration in Fig. 5 allows all adder inputs to be efficiently utilized and is most optimal when p is a power of 2.

The accumulating register is initialized with (signed 2's complement) -θ at every gamma cycle, which eliminates the need for any comparator for output spike generation. The $(log_2p+1)^{th}$ bit of the output indicates if the accumulated body potential has crossed the threshold and triggers a 3-bit counter to generates an 8-cycle wide pulse (output spike).

IV. STDP & R-STDP Implementation

STDP is a distinctive feature of TNNs. STDP learning is unsupervised and local to each synapse. It can perform inference and online continual learning at the same time. In this work, we propose an STDP design that is both effective in learning and implementable using standard CMOS technology.

A. Proposed STDP Update Rules

Our learning method is a customized version of the classic Spike Timing Dependent Plasticity (STDP) [1]. STDP is implemented locally at each synapse as shown in Fig. 6. The proposed STDP learning rules are summarized in Table I. Here, x(t) and z(t) represent input and output spiketimes respectively. Δw denotes change in weight and B(μ) represents a Bernoulli random variable with probability μ.

STDP update rules are divided into four major cases, corresponding to the four combinations of input and output

TABLE I: STDP Update Rules

Reward	Input Conditions	Weight Update
1	$x(t) \neq \infty$; $x(t) \leq z(t)$	$\Delta w = +B(\mu_{capture}) * max(F(w), B(\mu_{min}))$
	$z(t) \neq \infty$ $x(t) > z(t)$	$\Delta w = -B(\mu_{backoff}) * max(F(w), B(\mu_{min}))$
	$x(t) \neq \infty$; $z(t) = \infty$	$\Delta w = 0$
	$x(t) = \infty$; $z(t) \neq \infty$	$\Delta w = -B(\mu_{backoff}) * max(F(w), B(\mu_{min}))$
	$x(t) = \infty$; $z(t) = \infty$	$\Delta w = 0$
-1	$x(t) \neq \infty$; $x(t) \leq z(t)$	$\Delta w = -B(\mu_{capture}) * max(F(w), B(\mu_{min}))$
	$z(t) \neq \infty$ $x(t) > z(t)$	$\Delta w = 0$
	$x(t) \neq \infty$; $z(t) = \infty$	$\Delta w = +B(\mu_{search})$
	$x(t) = \infty$; $z(t) \neq \infty$	$\Delta w = 0$
	$x(t) = \infty$; $z(t) = \infty$	$\Delta w = 0$
0	$x(t) \neq \infty$; $x(t) \leq z(t)$	$\Delta w = 0$
	$z(t) \neq \infty$ $x(t) > z(t)$	$\Delta w = 0$
	$x(t) \neq \infty$; $z(t) = \infty$	$\Delta w = +B(\mu_{search})$
	$x(t) = \infty$; $z(t) \neq \infty$	$\Delta w = 0$
	$x(t) = \infty$; $z(t) = \infty$	$\Delta w = 0$

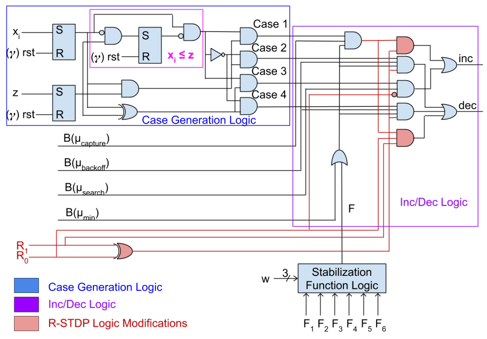


Figure 7: Implementation

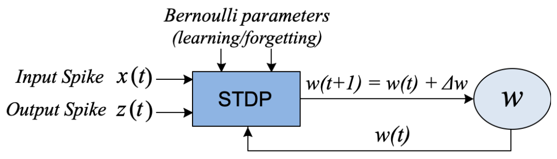


Figure 6: Local STDP Update Process

spikes (represented by x(t) and z(t) respectively) being present ($\neq \infty$) or absent ($= \infty$) . When both are present, two subcases are formed based on the relative timing of the input and output spikes in the classical STDP manner [1]. In effect, a synaptic weight is incremented (strengthened) if there is an input spike and it either contributed (Case 1) or can potentially contribute (Case 3) to the output spike; else it is decremented.

The STDP update function either increments the weight by Δw (up to a maximum of w_{max} = 7), decrements the weight by Δw (down to a minimum of 0), or leaves the weight unchanged. The Δ values (1, 0 or -1) are defined using Bernoulli random variables (BRVs) with parameterized learning probabilities denoted as B(μ) with a descriptive subscript. $F(w)$ is a stabilization function (=$B((w/w_{max})(1-w/w_{max}))$) which makes the weights "sticky" at both ends (0 and 7) [8], [9].

B. Proposed STDP Implementation

The proposed STDP logic implementation is shown in Fig. 7. It generates 2 control signals (increment/decrement) at the output that feed into the synaptic weight counters described in Fig. 4. Note that STDP updates (and the associated resets) are performed at the end of a computational cycle (or onset of next gamma clock); inputs for the new computational cycle begin a unit clock cycle later. The proposed STDP logic implementation can be partitioned into three components.

1) Case Generation Logic: The per-synapse case generation logic compares the synapse's input spiketime (x_i) with its post-synaptic neuron's output spiketime (z) and generates 4 control signals corresponding to the 4 cases in Table I. Case 5 is implicitly invoked when none of the other 4 cases is a 1. The logic equations implemented for the 4 STDP cases are:

- Case 1: $(x_i \leq z).(x_i).(z)$ • Case 2: $\overline{(x_i \leq z)}.(x_i).(z)$
- Case 3: $(x_i \leq z).(x_i \oplus z)$ • Case 4: $\overline{(x_i \leq z)}.(x_i \oplus z)$

Note that $((x_i \leq z))$ is implemented here using a much simpler *temporal* comparator as opposed to a binary comparator. If z arrives prior to x, the output is 0; else x is allowed to pass.

2) Stabilization Function Logic: This logic selects 1 BRV from a set of finite BRVs generated by $F(w)$, based on the synaptic weight. For w_{max} = 7, there are 6 non-zero BRVs to choose from. The output bit is generated by an 8-to-1 multiplexer controlled by 3-bit weight.

3) Inc/Dec Logic: The inc/dec logic assumes 4 BRV inputs from the LFSR network corresponding to the four STDP cases. The *max* operation in Table I is simply implemented by 'OR'ing 'F' with *min* BRV input. The output of the stabilization logic is used along with the cases from case generation logic to generate *inc* and *dec* outputs.

C. Proposed R-STDP Implementation

This subsection introduces a variation of the proposed STDP method capable of *reinforcement learning* (R-STDP) [18] that uses an external *reward* signal to drive its learning process in a desired direction. It involves three forms of reinforcement:

- When the column's (non-null) output matches the desired action, *reward* = '1'. It operates as per Table I; except case 3 results in no synaptic weight update.
- When the column's (non-null) output does not match the desired action, *reward* = '-1'. Only Cases 1 and 3 are performed; for Case 1, weight is actually decremented instead of incremented.
- When the column produces no output, i.e., no neuron spikes, *reward* = '0' and only Case 3 operates.

In effect, desired behavior is reinforced and undesirable behavior is repressed using a single *global* reward signal. Note that R-STDP is still applied locally to each neuron and is typically deployed in the final layer of a TNN. The logic modifications for R-STDP are minimal and straightforward as highlighted in Fig. 7. *reward* is a 2-bit signal (which encodes '-1', '0' and '1' as '11', '00' and '01' respectively). Unsupervised STDP is invoked when *reward* is '10'.

The implemented STDP/R-STDP learning rules are capable of performing extremely efficient online incremental learning (see Section VII). To the best of our knowledge, such gate-level hardware-efficient implementations of STDP/R-STDP rules for TNNs have not been presented or published before.

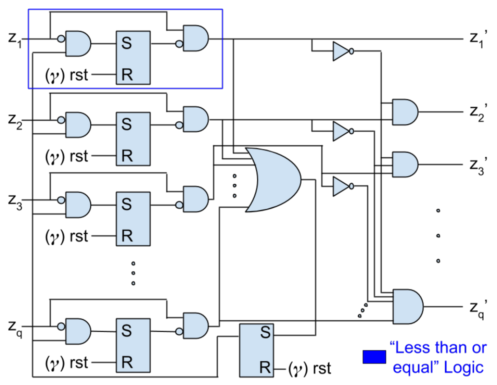


Figure 8: WTA Inhibition for a Column of q Neurons

V. Column Implementation

A column is a fundamental functional unit in TNNs [25], [26], much like ALUs in von-Neumann computers. As shown in Fig. 3b, a $p\times q$ column contains q excitatory neurons and a synaptic crossbar connecting the p inputs to the q neurons via $p\times q$ synapses. A column supports unsupervised learning via STDP or supervised learning via R-STDP, followed by WTA lateral inhibition to assist in weight convergence. A single column supported by STDP/R-STDP and WTA becomes a fully operational TNN, capable of performing online continual learning and inferencing. Columns can be used to create larger TNNs by stacking multiple columns to form a multi-column layer, and by cascading multiple layers into a multi-layer TNN.

Winner-take-all (WTA) inhibition is a distinctive feature of a column that selects the first spiking neuron and allows its output spike to pass through intact, while nullifying other neurons' outputs. Fig. 8 shows the logic diagram for 1-WTA inhibition across q neurons in a column. The inhibition operation is performed by a latch-based less-than-or-equal temporal comparison unit. The first spike is found through a large 'OR' gate, or a tree of small OR gates, (performing a temporal 'min' function) and is fed back through a latch which holds the signal at 1 until the next gamma cycle. Any input pulse coming to the latch after this signal is blocked, so only the first spikes are passed. Tie breaking is implemented by selecting the spiking neuron with the lowest index.

VI. Microarchitecture Framework Evaluation

Scalable neuron and column designs are implemented in System Verilog; synthesis results are generated using open-source 45nm Nangate standard cell library [12] and Synopsys tools. Hardware design is evaluated in terms of area (A), critical path delay (D), computation time (T) and power (P). T is the time taken to process one input (one *gamma* cycle).

A. Gate-Level Characteristic Scaling Equations

We derive characteristic scaling equations (Table II) for A, D (neuron), T (column) and P based on gate count ('AND' equivalents) and number of signal transitions, parameterized in terms of number of neurons (q) and number of synapses per neuron (p). The procedure is as follows: 1) Gate count is used as a surrogate for area and static power. 2) Number of gates in the critical path is used for D; T is derived using the *gamma* period, $T = 15 * D$. 3) Number of gate transitions is used for dynamic power. These equations can serve as a powerful tool for design space exploration, as they can help estimate the hardware complexity of arbitrary TNN designs.

From our gate-level analysis for a single neuron, synapses (including STDP) constitute almost 90% (50% synaptic FSM and 40% STDP logic) of the entire neuron complexity while the neuron body accounts for the remaining 10%. In a single column, neurons constitute almost the entirety of column complexity; WTA incurs negligible cost (less than 1%).

TABLE II: Characteristic scaling equations for A, D/T and P for a neuron with p synapses and a $p\times q$ column.

Metrics	**Neuron**	**Column**
A	$102p + 8log_2p + 36$	$102pq + 8qlog_2p + 44q + q^2$
D / T	$6log_2p + 4$	$90log_2p + 60$
P_{static}	$102p + 8log_2p + 36$	$102pq + 8qlog_2p + 44q + q^2$
$P_{dynamic}$	$204p + 185log_2p + 241$	$204pq + 185qlog_2p + 257q + 2q^2$

TABLE III: A, T and P (in 45 nm CMOS) for three column sizes of 64×8, 128×10, 1024×16, with STDP and R-STDP.

	Synapses x Neurons	**Gate Count**	**Area** [mm^2]	**Comp. Time** [ns]	**Power** [mW]
STDP	64 × 8	52K	0.05	29	0.3
	128 × 10	129K	0.13	32	0.6
	1024 × 16	1,639K	1.65	42	8.0
R-STDP	64 × 8	54K	0.05	29	0.3
	128 × 10	135K	0.14	32	0.7
	1024 × 16	1,721K	1.75	42	8.4

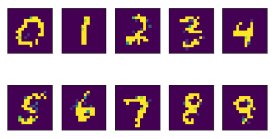
B. Post-synthesis Evaluation of Column Designs

Area, power and critical path delay are obtained directly from Design Compiler, and computation time is derived as earlier. We use the low power process corner for synthesis with operating frequency of 100 kHz and voltage of 0.95 V.

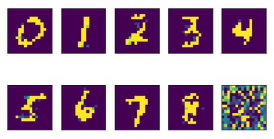
Table III presents 45 nm post-synthesis results for three column configurations for STDP and R-STDP learning rules: 1) a small 64×8 column; 2) a medium 128×10 column; and 3) a large 1024×16 column. The *gamma* cycle for the large 1024×16 column with around 1.7M gates is 42.3 ns (23.64 MHz). It has an area and power footprint of 1.65 mm^2 and 7.96 mW with STDP in 45nm, less than 1% of the area and power budget of typical mobile SoCs. Note that the overhead for R-STDP is minimal; it increases die area and power by only 5% relative to STDP while adding supervision to learning.

VII. Online Incremental Learning

In contrast to the typical epoch-based training with global back-propagation, STDP is an online local learning method that processes inputs in a streaming manner targeting online real-time applications. This section uses a subset of MNIST, with images resized from 28×28 to 16×16, to illustrate online incremental learning for TNNs. Because our focus is on online learning, the standard MNIST benchmark protocol for offline



(a) Trained for Digits 0 - 9



(b) Trained for Digits 0 - 8

Figure 9: Synaptic weight matrices converge to image centers resembling MNIST digits in 10,000 samples.

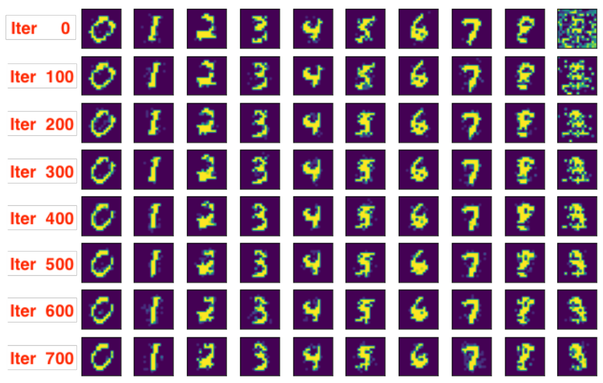


Figure 10: Online Incremental Learning: STDP learns a previously unseen input number '9' within 500 examples.

training/testing does not apply. From our experiments, using just a single (256×10) column and resized MNIST, several interesting capabilities of TNNs can be observed.

1) *Online Classification via Centroid Formation*: Fig. 9a shows the synaptic weights converged to the 10 class centroids via R-STDP, which resemble the corresponding digits. This shows the efficacy of R-STDP in driving the weights towards class centroids.
2) *Fast Training Convergence*: The synaptic weights in Fig. 9a and Fig. 9b converged after approximately 10,000 training samples, which implies that TNNs can learn very quickly and can generalize from small datasets.
3) *Online Incremental Learning*: In this experiment, supervised R-STDP training is first performed with only 9 classes (0 to 8) by hiding the digit '9', resulting in the converged weights shown in Fig. 9b. Then the digit '9' is introduced in the input sequence without labels to illustrate the ability to dynamically learn a previously unseen class in an unsupervised fashion. As shown in Fig. 10, the synaptic weights converge to the digit '9' after only about 500 testing samples via STDP.

Thus, online incremental learning enables a TNN to adapt to new input data not seen before during the original (offline) training. Continual learning allows a TNN to keep learning and improving its performance concurrently with inference.

VIII. Concluding Remarks

Previous works in [3], [5], [8], [18], [25], [26] have shown that TNNs can achieve online brain-like sensory processing and learning for vision and time-series applications. This work proposes a microarchitecture framework for directly implementing arbitrary TNNs using the building blocks: neurons, columns and online STDP/R-STDP learning. This work demonstrates the hardware implementation feasibility of TNNs using off-the-shelf CMOS technology and design tools, and represents an initial step in a promising direction for future research. The implementation results in this work should be viewed as a first opportunistic attempt, using existing design methods and tools. There are promising innovations, including custom macro cells and novel synthesis tools, that can be developed to further enhance the proposed design framework.

References

[1] G. Bi and M. Poo, "Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type," *Journal of neuroscience*, vol. 18, 1998.
[2] S. Bohte, J. Kok, and H. La Poutre, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, 2002.
[3] S. Chaudhari, H. Nair, J. M. Moura, and J. P. Shen, "Unsupervised clustering of time series signals using neuromorphic energy-efficient temporal neural networks," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 7873–7877.
[4] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in computational neuroscience*, vol. 9, p. 99, 2015.
[5] M. Dong, X. Huang, and B. Xu, "Unsupervised speech recognition through spike-timing-dependent plasticity in a convolutional spiking neural network," *PloS one*, vol. 13, no. 11, p. e0204596, 2018.
[6] P. Fries, D. Nikolić, and W. Singer, "The gamma cycle," *Trends in neurosciences*, vol. 30, no. 7, pp. 309–316, 2007.
[7] R. Guyonneau, R. VanRullen, and S. J. Thorpe, "Neurons tune to the earliest spikes through stdp," *Neural Computation*, vol. 17, 2005.
[8] S. R. Kheradpisheh, M. Ganjtabesh, and T. Masquelier, "Bio-inspired unsupervised learning of visual features leads to robust invariant object recognition," *Neurocomputing*, vol. 205, pp. 382–392, 2016.
[9] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, "Stdp-based spiking deep convolutional neural networks for object recognition," *Neural Networks*, vol. 99, pp. 56–67, 2018.
[10] S. Kim, S. Park, B. Na, and S. Yoon, "Spiking-yolo: spiking neural network for energy-efficient object detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020.
[11] W. Kistler, W. Gerstner, and J. Hemmen, "Reduction of the hodgkin-huxley equations to a single-variable threshold model," *Neural computation*, vol. 9, 1997.
[12] J. Knudsen, "Nangate 45nm open cell library," *CDNLive, EMEA*, 2008.
[13] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, 2015.
[14] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in neuroscience*, 2016.
[15] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.
[16] T. Masquelier and S. J. Thorpe, "Unsupervised learning of visual features through spike timing dependent plasticity," *PLoS computational biology*, vol. 3, 2007.
[17] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 7, pp. 3227–3235, 2017.
[18] M. Mozafari, M. Ganjtabesh, A. Nowzari-Dalini, S. J. Thorpe, and T. Masquelier, "Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks," *Pattern Recognition*, vol. 94, 2019.
[19] T. Natschläger and B. Ruf, "Spatial and temporal pattern analysis via spiking neurons," *Network: Computation in Neural Systems*, 1998.
[20] OpenAI, "AI and Compute," https://openai.com/blog/ai-and-compute/.
[21] B. Parhami and C.-H. Yeh, "Accumulative parallel counters," in *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, vol. 2. IEEE, 1995.
[22] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai. corr abs/1907.10597 (2019)," *arXiv preprint arXiv:1907.10597*, 2019.

[23] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, “Going deeper in spiking neural networks: Vgg and residual architectures,” *Frontiers in neuroscience*, vol. 13, p. 95, 2019.
[24] J. E. Smith, “Space-time algebra: A model for neocortical computation,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.
[25] J. E. Smith, “A neuromorphic paradigm for online unsupervised clustering,” *arXiv preprint arXiv:2005.04170*, 2020.
[26] J. E. Smith, “A temporal neural network architecture for online learning,” *arXiv preprint arXiv:2011.13844*, 2020.