Spiking Neuromorphic Networks for Binary Tasks

James S. Plank
ChaoHui Zheng
jplank@utk.edu
czheng4@utk.edu
University of Tennessee
Knoxville, TN, USA

Catherine D. Schuman
schumancd@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Christopher Dean
chrisdean258@gmail.com
University of North Carolina
Chapel Hill, NC, USA

ABSTRACT

In this paper, we focus on the hand construction of small-scale, spiking, neuromorphic networks. They are partitioned into two sets. The first set performs the binary operations AND, OR and XOR. For each of these operations, there are eight scenarios that we consider, with four types of encodings of binary values to spikes, and neurons that feature or prohibit leak. The second set of networks perform conversions of binary values from one type of encoding to another, again considering both leak or no leak. These networks are presented graphically and with spike-raster plots that illustrate their activity. We tabulate metrics of interest concerning the size, activity and speed of these networks. Our goal with this work is to enable the composition of multiple spiking neural networks, perhaps trained with other methodologies, without requiring information to leave a neuroprocessor for processing by conventional hardware.

CCS CONCEPTS

• **Hardware** → *Neural systems*; • **Theory of computation** → **Design and analysis of algorithms**.

KEYWORDS

spiking neural networks, binary operations, spike encodings

ACM Reference Format:
James S. Plank, ChaoHui Zheng, Catherine D. Schuman, and Christopher Dean. 2021. Spiking Neuromorphic Networks for Binary Tasks. In *International Conference on Neuromorphic Systems 2021 (ICONS 2021), July 27–29, 2021, Knoxville, TN, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3477145.3477168

1 INTRODUCTION

Recurrent, spiking neuromorphic networks (RSNNs) have held promise for their brain-like computing ability. Despite the numerous successes in applying these networks to a variety of specific problems, a glaring gap exists in our knowledge of how to design networks to solve general problems. The most successful approaches to date have applied backpropagation to these systems so that approaches that are successful in conventional neural networks may also be leveraged for spiking systems [21, 22]. There have been other promising training approaches, such as reservoir computing [12], STDP [13], genetic algorithms [17] and localized backpropagation [5]. However, no general and successful methodology has emerged to train RSNNs.

There is a separate body of work where these networks are designed by hand, tailoring the timing of the spikes to perform exact mathematical functions or discrete graph algorithms. Examples include factorization [16], shortest path calculations [10], sorting and median calculations [24]. These approaches are in direct opposition to those that train or use machine learning, and expose themselves to a number of criticisms. For example:

- They do not attempt to mimic the brain in any way.
- They don't "learn".
- They are often rigidly timed and inflexible.
- Conventional hardware has been tuned to implement these tasks with very high performance.

While acknowledging these criticisms, we point out that these smaller-scale theoretical works demonstrate what RSNNs *can* do definitively and quantitatively, without relying on empirical or statistical evaluations of effectiveness.

This paper works on a smaller scale, utilizing a general integrate-and-fire model of RSNN with optional leak. We present RSNNs that implement two functionalities.

(1) The binary operations AND, OR and XOR.
(2) Conversion from one binary input encoding to another.

We are flexible with respect to how inputs and outputs are encoded. Specifically, we use direct, binned, frequency-based and temporal input encodings, providing networks for each as they apply to binary problems.

Our intent with this work is two-fold. First, by focusing on the very small scale, we hope to provide some intuition about how RSNNs perform basic functions. Second, the networks presented herein can compose building blocks for other work in RSNNs. The functionalities presented here are often implemented by conventional processing elements in neuromorphic systems (e.g. by general processing cores on Intel's Loihi [7]), and implementing them using neuromorphic computing elements will improve the functionality, speed and energy consumption of the neuromorphic system.

2 RELATED WORK

Though the vast majority of algorithmic and application work in neuromorphic computing has focused on training or learning approaches [19], in recent years, there has been an increase in interest in utilizing neuromorphic computers for non-neural network

ICONS 2021, July 27–29, 2021, Knoxville, TN, USA

ACM ISBN 978-1-4503-8691-3/21/07…$15.00
https://doi.org/10.1145/3477145.3477168

tasks [1]. Different properties of neuromorphic computers have been exploited to achieve computational performance on these non-neural network tasks; however, utilizing these characteristics of neuromorphic systems effectively often requires rethinking how a problem is defined and understanding how to effectively build the corresponding spiking neural network to address the task.

Research into massively parallel computation using neuromorphic networks has led to exact network construction that does not rely on machine learning. Monaco and Vindiola exploit massive parallelism and constant time synaptic integration to implement a constant time check for smoothness in integer factorization [16]. Graph algorithms such as shortest path and minimum spanning tree as implemented on neuromorphic computers also exploit the massively parallel nature of computation [10], and by exploiting these characteristics, it has been shown that neuromorphic computers have a provable computational advantage on shortest path calculations [2]. Additionally, large-scale parallel computation in neuromorphic systems have also been exploited for sorting and median calculations [24].

Several groups have exploited the massively parallel nature of neuromorphic computers, along with inherent stochasticity properties on many neuromorphic implementations to implement neuromorphic solutions to constraint satisfaction tasks [9, 15, 25]. The stochasticity of neuromorphic systems has also been utilized to allow for implementing Markov random walks [20], which have in turn been utilized to perform partial differential equation solving [23].

The computational capabilities of certain types of neural computation have been established previously. Maass established that spiking neural newtorks with synaptic plasticity are Turing complete in 1996 [11]. Recently, Cabessa established that rational-weighted recurrent neural networks employing spike-timing-dependent-plasticity (STDP) are Turing-complete [6] as well. Though these works establish that certain subsets of neuromorphic computers (i.e., those with the properties described) are Turing complete, neither gives the best indication of *how* to implement general computation onto neuromorphic computers via RSNNs.

There have been efforts to define programs of neuromorphic computers via composition of higher level abstractions, such as Fugu [3]. Fugu specifically focuses on defining a common intermediate representation that can be used across hardware backends and algorithms. However, there has been relatively little analysis focused on the best practices associated with manually programming neuromorphic computers.

3 RSNN MODEL

We assume the following model of RSNN. A neural network is a graph, where neurons compose the nodes and synapses are directed edges between neurons. Neurons have configurable thresholds and synapses have configurable weights and delays. Neurons follow the "integrate and fire" methodology. By that, a neuron's central feature is its *potential*, which is simply a numeric value. The potential starts at a base value, which in this paper we will assume to be zero. The potential's value may be increased or decreased by fire events on incoming synapses. Upon a synapse fire (sometimes called a spike), the synapse's weight is added to or subtracted from the post-synaptic neuron. This process is called *integration*. If a neuron's potential reaches or exceeds its threshold, the neuron will fire (also called spiking), which will trigger each of its outgoing synapses to fire on its post-synaptic neuron after the synapse's delay. Additionally, after firing, the neuron resets its potential. To be precise, a RSNN is defined by the graphical description of its neurons and synapses, along with a setting of the neuron thresholds, synapse weights and synapse delays.

Neurons partition time into integration intervals. During an integration interval, a neuron's incoming synaptic fires modify its potential up or down, but the neuron does not fire unless its potential reaches or exceeds its threshold at the end of the interval. This is often an important consideration when designing RSNNs, because it obviates concern about the precise arrival times of synaptic fires within an interval. In this work, we assume that integration intervals are one cycle in duration and end on the integer value, at which point neurons fire if their potential is at least equal to their thresholds.

Some RSNN systems contain *leak*, whereby neuron potentials "settle" to their base values over time. Some also contain *plasticity*, where synapse weights self-adjust according to their impact on their post-synaptic neurons. We do not consider plasticity in this paper, but we do consider leak. In particular, we either disallow leak (which we call *"no-leak"*) , or we assume that neurons leak all of their charge at the end of each integration cycle (*"leak"*). In *leak* networks, neurons start at their base charge values at the beginning of each integration cycle, regardless of whether they fire in that cycle.

We specify a precise example of an RSNN in Figure 1. The neurons Y and Z both have thresholds of $T = 1$, and all potentials start at zero. Suppose that at time zero, neuron X fires. This causes the XY and XZ synapses to fire at time 1. Since integration intervals end on integral values, at time 1, neuron Y's potential will be 1, and neuron Z's potential will be 0.5 (this assumes *no-leak*). Accordingly, neuron Y fires at time one. This causes the YZ synapse to fire at time 2, raising neuron Z's potential to 1.25. As that exceeds neuron Z's threshold, at time 2 it fires. This activity is demonstrated by the spike raster plot at the right of Figure 1, which depicts the neurons' potentials at each timestep. A potential of 0.5 is denoted in blue, and a neuron's fire is denoted in black.

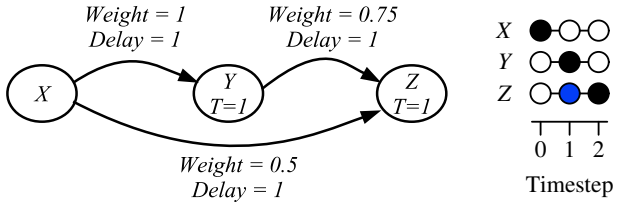

Figure 1: A simple RSNN to illustrate integration and firing, plus a spike raster plot that results when X spikes at timestep 0.

To interact with its environment (or potentially with other RSNNs), a subset of the RSNNs neurons is designated as *input* neurons, and

another, potentially overlapping subset as *output* neurons. Whatever entity controls the environment may apply fire events with specific times and weights to the input neurons individually, and may record the times that output neurons fire.

This RSNN model encompasses most neuroprocessors that support RSNNs. There are additional constraints that some neuroprocessors enforce, such as a limited range of potential values (such as -128 to 127) or connectivity restrictions.

4 INPUT AND OUTPUT ENCODINGS

There is no consensus on the "best" way to encode information for a RSNN and from an RSNN. In this section we describe some of the more popular encodings that researchers have employed.

4.1 Input Encodings

Direct: Here the input values are converted directly to the weights that are applied to the input neurons. Obviously, floating point values, or values whose domains do not match the domains of neuron potentials, may be scaled and discretized. Mathematically, if the input domain is from D_{min} to D_{max}, then an input value of v results in a single spike, applied at time 0, with a weight of $f\frac{v-D_{min}}{D_{max}-D_{min}}$, where f is a scaling factor that takes into account the constraints of the neuroprocessor that is executing the neural network.

Population Coding / Bins: With population coding, multiple neurons are dedicated to a single input, and the determination of which neuron fires as a result of an input value is made by a special-purpose function [8]. While functions exist of varying complexity, the most straighforward mapping of inputs to neurons is a linear mapping, where the input domain is partitioned into equal size regions, and one neuron is assigned per region. We call the neurons *bins*. Each input value is assigned to a bin and results in a spike being applied to the input neuron that corresponds to the bin. If there are b bins, then the bin number (zero-indexed) is computed with $\lceil b\frac{v-D_{min}}{D_{max}-D_{min}}\rceil$, Besides the mapping function, there are many options with respect to the weights and timings of the spikes within the bins [18]. In this paper, we make the simplest assumption that each value will result in exactly one spike of weight one, at time zero in its appropriate input neuron.

Rate Coding / Spike Counts / Stochastic Logic: These are input encoding techniques where a single neuron is used for an input, and the magnitude of the input maps to the number of spikes on the neuron. In general, larger inputs map to more spikes; however, as with population coding, there are many ways to perform this mapping, which may involve thresholds, filters, error functions and stochasticity [8]. In this paper, we take a simple approach called *spike count* encoding [18]. With spike counts, a value is converted into a number of spikes, and these spikes are generated at a prespecified interval, such as one spike per unit time, regardless of the number of spikes. Specifically we specify the value M to be the maximum number of spikes for an input value. Then, $\lceil M\frac{v-D_{min}}{D_{max}-D_{min}}\rceil$ spikes are applied for an input of v. Note how this is similar to binning, where the M is used in place of b.

Temporal encoding: A fourth encoding technique is *temporal*, where an input value is mapped into one or more spikes that are applied to a single neuron, and the magnitude of the value is converted into the timing of the spike(s). Systems like Slayer [22] employ temporal encodings to yield differentiable functions for backpropagation. As with population and rate coding, there have been many mapping functions defined in the literature with various parameterizations. And like the other techniques, we take a simple, linear-scaled view of temporal encoding: a value v is encoded into a spike time t_v relative to a reference time t such that $t_v = t - fv$, where f is a scaling factor.

4.2 Output encoding

There is no analog to **Direct** input encoding, unless the outputs are binary. In that case, a spike corresponds to an output of one, and no spike corresponds to an output of zero. For non-binary outputs, there is no direct output encoding, because neurons do not change their values when they spike. Each of the other encoding techniques outlined above have corresponding output encodings.

Bins: As with inputs, one may allocate multiple neurons to a single output, and then assign each neuron to be a "bin" that corresponds to a specific output or range of outputs. One difference from input encoding is that there is nothing to prevent a network from spiking into multiple output bins. In that case, the simplest solution is to use the output bin that has the most spikes. There are other solutions, such counting the spikes and applying them to a softmax activation function, or using a dot product with a "key" vector [21].

Spike Counts / Rate Coding / Stochastic Logic: With these techniques, only one output neuron is employed. The output spikes are counted and scaled appropriately to construct an output value.

Temporal decoding: With temporal decoding, the most recent output spike is converted to a value according to its time relative to a reference time.

4.3 Summary

To summarize, in this paper we consider the following encoding techniques: direct, bins, spike counts and temporal. They are all considered in their simplest forms, with linear scaling and no additional hyperparameters. We consider the same decoding techniques, noting that direct decoding is posible because the outputs are binary.

5 RSNNS FOR BINARY OPERATIONS

In this section, we consider three very simple binary computations: AND, OR, AND_NOT and XOR. This section is rather large, as we consider *no-leak* and *leak* networks, plus all of the input encodings enumerated above along with their analogous output encodings. As such, we present networks for 3x4x2 = 24 scenarios.

As in the Introduction, we acknowledge that RSNNs are typically explored for brain-inspired and machine-learning applications, and not precise binary calculations which are implemented more natually in CMOS. However, besides providing insights into how RSNNs compute, these small networks may be used for various tasks at the edges of, or as the glue between, larger, more complex RSNNs. For example, there are applications where multiple, independently trained RSNNs may be employed for different phases of the application, and the solution to managing these networks is to have an

external microprocessor keep track of the phase, and then direct the neuroprocessor to load a new RSNN when phases change [4].

With these binary building blocks, one may keep all RSNNs loaded on the neuroprocessor, and compose them so that their outputs are inputs to networks like the ones in this section, thus maintaining the network composition on the neuroprocessor. It is well-known that the most expensive part of running a neuroprocessor, from both the time and energy perspective, is managing its external connections and interactions. When used as glue, these networks obviate the need to leave the RSNN, saving on time and energy.

In Figure 2, we provide a legend for all of the drawings that follow. We represent the weights of synapses with lines of specific colors, and delays with specific line types. In all cases, neuron thresholds are equal to one.

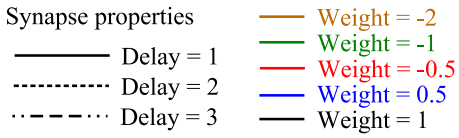

Figure 2: Legend of synapse weights and delays in the network figures.

5.1 Direct Encoding

With all of these binary tasks, there are two inputs which we term A and B. With a direct encoding, no spikes are generated for zero inputs, and when an input is one, a single spike with a weight of one is applied at time 0. The output is a single spike when the answer is one, and no spikes when the answer is zero. The timing of the spike depends on the network.

When neuron potentials are leaked to their base values (here zero) at the beginning of every integration cycle, and the applications involved do not require any "memory" over time, the resulting RSNNs can be quite simple. In Figure 3, we show the RSNNs for AND, OR and XOR. The AND and OR networks are very simple. The XOR network is constructed from two hidden neurons that calculate the AND_NOT of A and B, and of B and A respectively. Their OR is calculated, which results in the XOR of A and B.

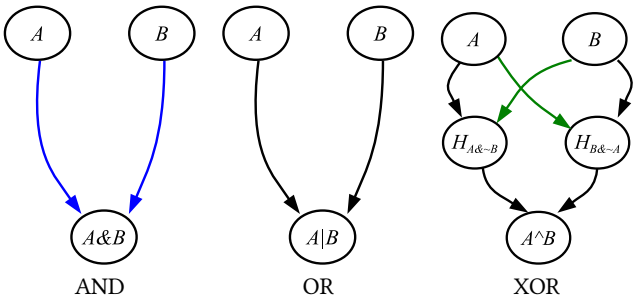

Figure 3: Networks to perform AND, OR, AND_NOT, and XOR with direct encodings and leak.

In Figure 4, we show spike-raster plots of the networks in the three cases of $[A, B] \in \{[0,0], [1,0], [1,1]\}$. We do not show $[0,1]$, because it is equivalent to $[1,0]$. The colors of the spike raster plots match the colors of the synapses. We use gray to denote when postive and negative charges cancel each other, resulting in a neuron potential of zero. We employ red boxes in the spike raster plots to show the input and output spike times. For example, all operations have their inputs at timestep 0. AND and OR record their output spikes at timestep 1, while XOR records its output spikes at timestep 2.

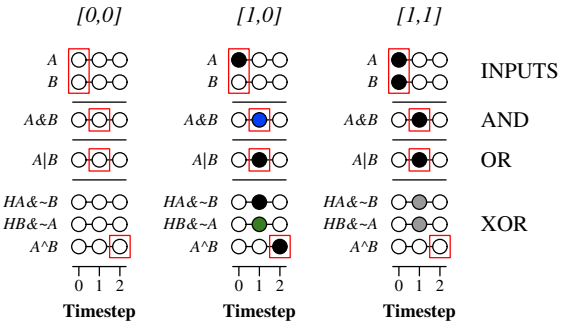

Figure 4: Spike raster plots of the networks in Figure 3, for $[A, B] \in \{[0,0], [1,0], [1,1]\}$. There is no plot for $[0,1]$ because the graphs are symmetric, and this case is equivalent to $[1,0]$. A gray circle means that positive and negative charges canceled to zero on this timestep.

There are many features of these networks that are worth considering. Obviously, the number of neurons and synapses represent their sizes, but there are other metrics of interest, such as spiking activity and time-to-output, which impact power consumption and speed. We tabulate these and other metrics of interest for all of these network in Section 6.

When there is no leak, the neurons retain their potentials from cycle to cycle. The networks for AND and XOR must add some complexity to eliminate residual neuron potentials that were conveniently leaked away in RSNNs in Figure 3. In Figure 5, we show these networks. There is no network shown for OR because the network in Figure 3 also computes OR with no leak. We show the networks' spike raster plots in Figure 6. We omit OR, as it is identical to Figure 4.

The major addition for leak is in the AND network, where the $A\&B$ neuron cancels the potentials of H_A and H_B when both A and B spike. If only one of them spikes, then its corresponding H neuron cancels the 0.5 potential stored in the $A\&B$ neuron. The XOR network is identical to the *leak* network, except that when a hidden neuron spikes, it cancels the negative charge in the other hidden neuron that was leaked away in Figure 3.

5.2 Encoding and decoding with bins

With bins, exactly one of the A neurons and exactly one of the B neurons spikes at time zero. Moreover, when we compute the binary function f, we need to have two output neurons — one that spikes when f is true, and the other that spikes when f is false. These

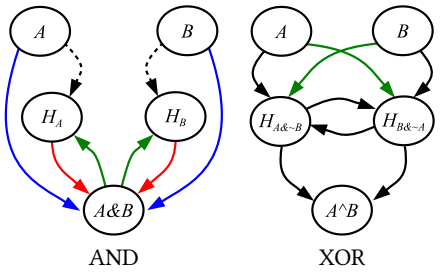

Figure 5: Networks to perform AND and XOR with direct encodings and no leak. The OR network in Figure 3 also calculates OR with no leak.

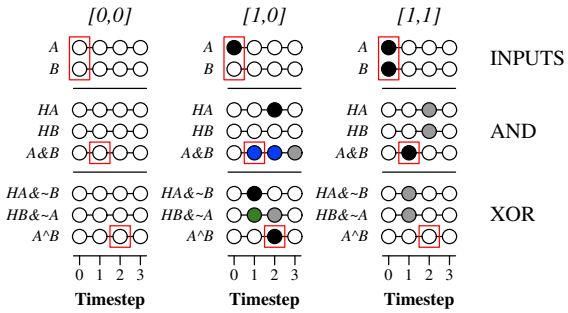

Figure 6: Spike raster plots of the networks in Figure 5.

constraints allow us to build some generic networks that apply to multiple binary functions, which we depict in Figure 7.

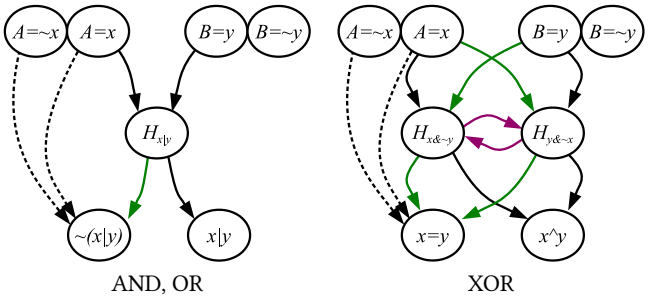

Figure 7: Generic networks to implement binary operations with binned inputs and outputs. The left network applies to both *leak* and *no-leak*. In the right network, the purple arrows are deleted with *leak*, and included with *no-leak*

.

We first consider the network on the left. In that network, we treat the combination of the two A neurons as a bias — since exactly one of them fires at time zero, we are guaranteed that the $\tilde{}(x|y)$ neuron always receives one unit of charge at time two. We then take the OR network from Figure 3 and turn its output into a hidden neuron $H_{x|y}$. It fires when either $A = x$ or $B = y$. This causes the $x|y$ output to fire at timestep 2, and cancels the bias charge going to $\tilde{}(x|y)$. When neither $A = x$ nor $B = y$, the bias causes $\tilde{}(x|y)$ to fire at timestep two.

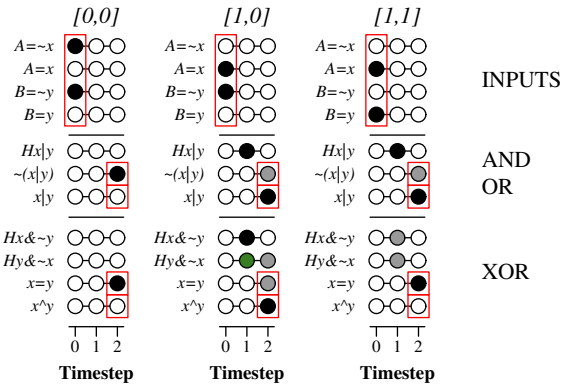

Figure 8: Spike raster plots of the networks in Figure 7.

In the right network, we apply the bias concept to the networks from Figure 3 (*leak*) and Figure 5 (*no-leak*). When x equals y, neither of the hidden neurons fire, and thus the x=y neuron fires at timestep two. Otherwise, exactly one of the hidden neurons fires, causing the $x\hat{}y$ neuron to fire at timestep two. The magenta synapses in the XOR network are required when there's no leak. If there's leak, they should be removed from the network.

We show the spike raster plots for the *no-leak* networks in Figure 8. The plots are identical when there is leak, except with input [1, 0], the neuron for $y\&\tilde{}x$ simply leaks its value from timestep 1 to timestep 2.

5.3 Spike count encoding

For binary applications, the spike count representation of zero is a single spike at time zero, and the representation of one is composed of two spikes, at times zero and one. Interestingly, the AND and OR networks of Figures 3 and 5 work with spike counts, unmodified. The reason is that both zero and one inputs start with a spike at time 0, and thus these networks *always* generate the first output spike. The second output spike is determined by the AND or OR function.

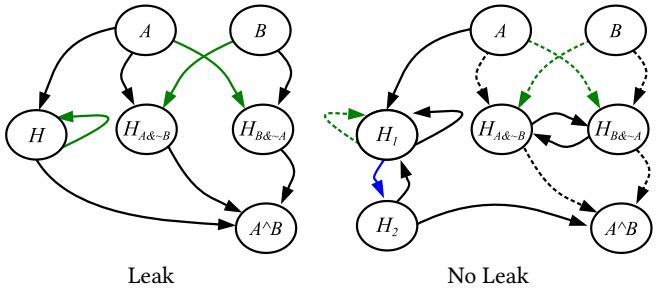

Figure 9: Networks for XOR and spike counts.

For XOR, we show networks for *leak* and *no-leak* in Figure 9, with spike raster plots in Figure 10. In Figures 10- 13, the red boxes

are two timesteps in width, because the inputs and outputs are measured over two timesteps, rather than one.

Unlike AND and OR, the first spikes of A and B do not cause the output to spike (because 1 XOR 1 equals zero). Thus, in these networks, we add hidden neurons to ensure that there is an initial spike, at time 2 in the *leak* network, and 4 in the *no-leak* network. With leak, there is one hidden neuron, H, that spikes with the first spike of A, and does not spike again, regardless of whether A spikes a second time. If A does not spike a second time, then its value leaks from -1 to zero at timestep three.

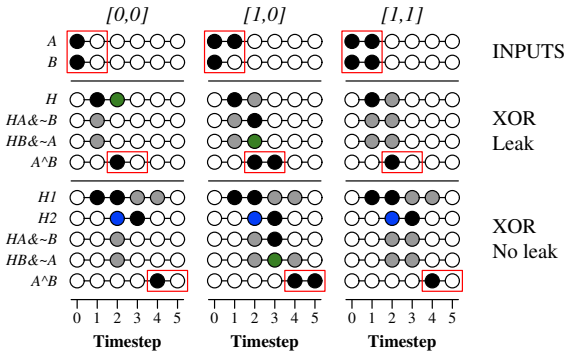

Figure 10: Spike raster plots of the networks in Figure 9.

In the *no-leak* network, neuron H_1 always spikes twice, which means that H_2 always spikes once. That generates the first spike. The second spike comes from the XOR network, which is equivalent to the network in Figure 5.

5.4 Temporal encoding

A simple temporal encoding for binary values is to choose an input reference time of 1 and a scaling factor of 1, so that zeros translate into a spike at time 1, and ones translate to a spike at time 0.

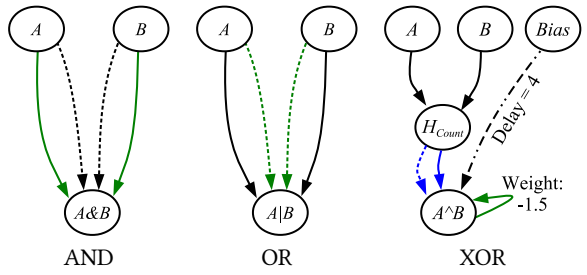

Figure 11: Temporal encoding networks with leak.

In Figure 11, we show networks that calculate AND, OR and XOR with temporal encoding and leak. If the neuroprocessor allows synapses that go to input neurons, one may insteada have each inhibitory synapse in the OR network go to the other input neuron with a delay of one. The XOR network uses a bias neuron that always fires at timestep zero. It uses the hidden neuron to convert the problem to a spike count encoding problem, where two spikes cause the output neuron to fire and cancel the bias, whereas one spike simply leaks away and allows the bias to spike the output. It is interesting that the AND and OR networks do not need a bias neuron, but XOR does — otherwise the network cannot differentiate the two cases when $A = B$.

The spike-raster plots for these networks are in Figure 12.

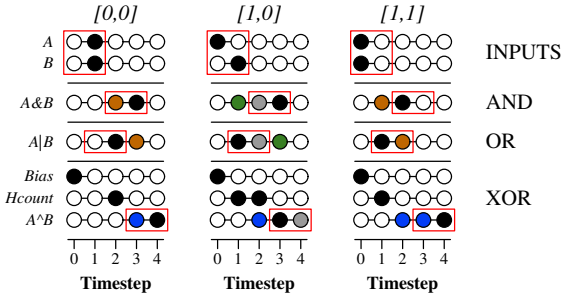

Figure 12: Spike raster plots of the networks in Figure 11.

When leak is disabled, it is surprising that the very simple network in Figure 3 calculates binary AND. The fact that there is always exactly one spike on A and B means that the output neuron AB will always fire, and it fires when it receives the later of the two spikes. The other binary operations are depicted in Figure 13 with spike rasters in Figure 14.

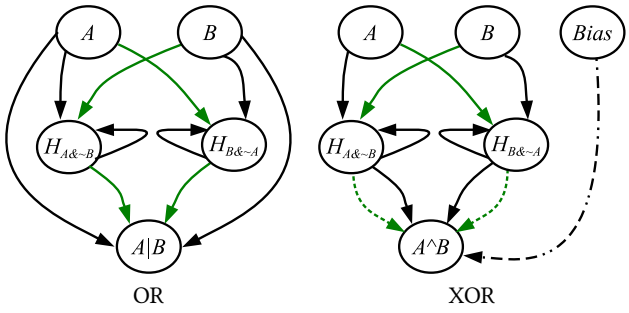

Figure 13: Temporal encoding networks without leak.

As in the *leak* networks, both of these networks operate by isolating $A\&\tilde{}B$ and $B\&\tilde{}A$, effectively separating the cases where A and B equal each other, and when they do not. With OR, the hidden neurons only fire when A and B fire at different times, and their function is to cancel the later of the A and B spikes. With XOR, exactly one of the hidden neurons fires at timestep 2, when A XOR B equals one. When this happens, the bias is canceled at timestep 3. When A equals B, neither of the hidden neurons fires, and the bias causes an output spike at timestep 3.

6 METRICS

As mentioned above, there are features of these networks that are of interest, in terms of size, activity (which relates to power consumption) and speed. For all of the networks depicted above, we tabulate the following metrics:

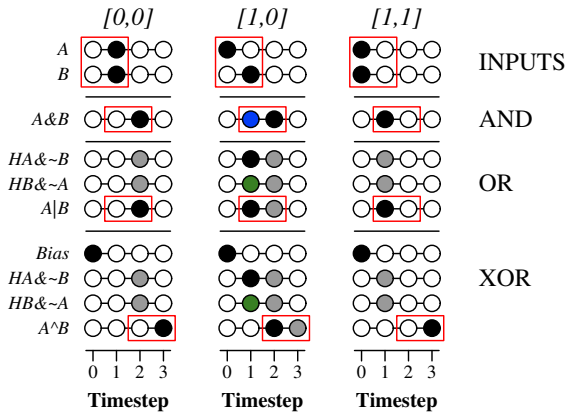

Figure 14: Spike raster plots of the networks in Figure 13.

- N: The number of neurons, including inputs and outputs.
- S: The number of synapses.
- A: The average activity of the network. This is measured by the average number of fires over the four potential combinations of A and B.
- O: The timestep where output begins. This is a measure of the speed of the network.
- R: The timestep where the network may be reused for another calculation. This is a measure of the throughput of the network, which may be different from the speed. For example, all of the networks in Figure 3 may be reused at timestep 1, even though their outputs don't occur until timesteps 1 or 2. Thus, these networks can perform more operations per unit time than, for example, the AND network in Figure 5, which cannot be reused until timestep 2.

These metrics are tabulated in Table 1. We use the following single-letter abbreviations for the encodings: D for direct, B for bins, S for spikes and T for temporal.

Encoding/	Leak					No Leak				
Operation	N	S	A	O	R	N	S	A	O	R
D-AND	3	2	1.25	1	1	5	8	1.75	1	2
D-OR	3	2	1.75	1	1			Same		
D-XOR	5	6	2.00	2	1	5	8	2.00	2	1
B-AND	7	6	3.75	2	1			Same		
B-OR	7	6	3.75	2	1			Same		
B-XOR	8	10	3.50	2	1	8	12	3.50	2	1
S-AND	3	2	4.25	1	2	5	8	4.75	1	3
S-OR	3	2	4.75	1	2			Same		
S-XOR	6	9	6.00	2	2	7	14	8.00	4	3
T-AND	3	4	3.00	2	3	3	2	3.00	1	2
T-OR	3	4	3.00	1	3	5	10	3.50	1	2
T-XOR	5	6	3.50	3	3	6	11	4.50	2	2

Table 1: Metrics of interest for the binary operator networks.

The table highlights some interesting properties of the encodings. First, the direct encodings have the smallest, fastest and most efficient networks. Next, the binned encodings feature low activity and quick reuse, even when the networks are larger (for example, the XOR networks). The spike encodings result in networks with greater activity, and slower reuse. The temporal networks feature low activity and quick output, but poorer resue than direct or binned encodings.

In every case but temporal encoding, the *leak* networks are equal to or better than their *no-leak* counterparts in every metric. With temporal encoding, the *no-leak* networks have equal or better speed and reuse than their *leak* counterparts, although in general their size and activities are larger.

7 CONVERSIONS

A second important functionality is to convert from one encoding to another. This allows one to modularize neuromorphic computing tasks by training subnetworks with different encodings, and gluing them together with these conversion modules. In this section we demonstrate networks that perform conversions from each type of encoding to the other. There are four encoding techniques, which we denote by single letters used in Table 1. There are 4x3 = 12 combinations of encoding techniques, each with *leak* and *no-leak* variants, for a total of 24 conversions. We denote them as $f \rightarrow t_l$, where f is the source encoder, t is the target encoder, and $l \in \{Y, N\}$ to denote leak.

The conversions from bins to the others are trivial to construct, and we do not show them. We draw the rest in Figures 15 through 17, with spike rasters in Figure 18.

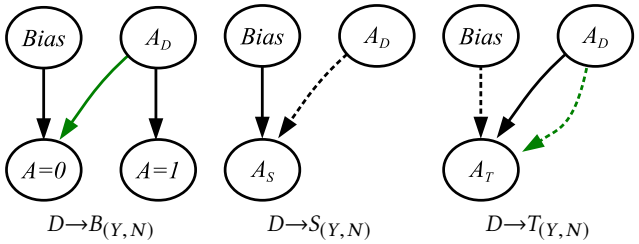

Figure 15: Conversions from the direct encoding to the other encodings.

The conversions from direct encoding require a bias neuron, because a zero input is represented by the lack of a spike. The bias allows us to always have a spike for zero. When converting to bins, the bias goes to the $A = 0$ bin, and when in input is one, the spike from A_D cancels the bias. The other two conversions in Figure 15 are straightforward, and since they don't require a neuron to carry a value from one timestep to another, they work in both *leak* and *no-leak* situations.

In Figure 16, we show networks that convert from spike encoding to the others. The top row of networks work with leak, and each of them leverages two synapses from the input neuron, one with a delay of one, and one with a delay of two. Both synapses have a weight of 0.5. Thus, when the input is zero, represented by a single spike, the synapses do not cause any firing. When the input

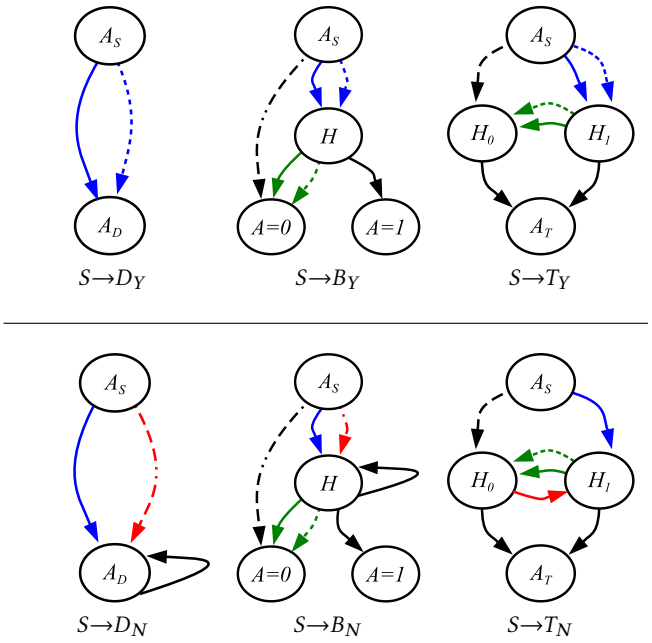

Figure 16: Conversions from binary spike encoding to the other encodings.

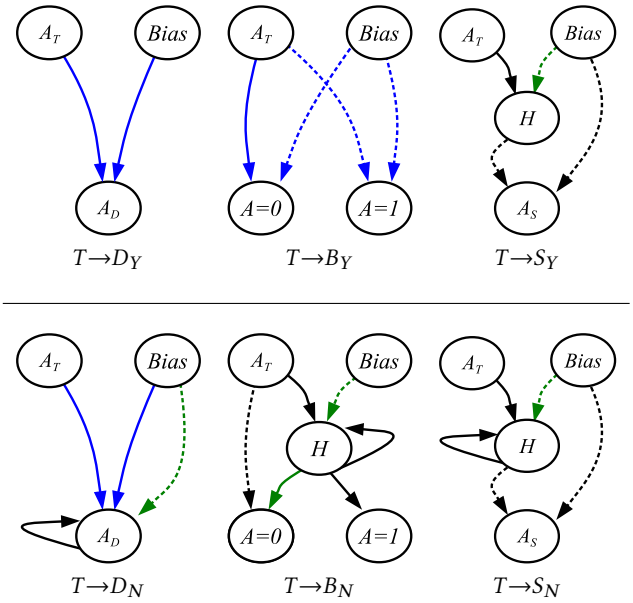

Figure 17: Conversions from temporal encoding to the other encodings.

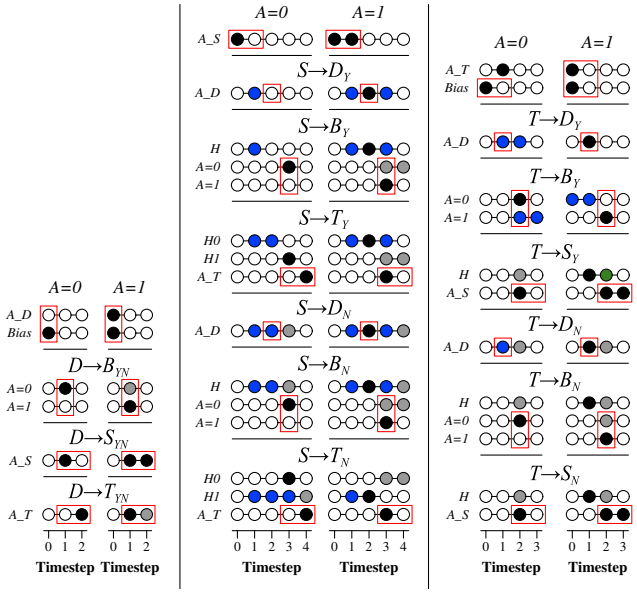

Figure 18: Spike-raster plots for the conversion networks.

is one, represented by two spikes, the synapses combine to spike at timestep 2. The three networks differ in how they combine that spike with other synapses so that zero and one are converted to their proper encodings.

The bottom row of networks are for *no-leak*. All three networks feature a single synapse of weight 0.5 coming from the input neuron. Since neuron potential is retained over timesteps, this synapse causes a fire when the input is one (two spikes). The networks are more complex, because any residual potential must be canceled rather than leaked away.

The networks in Figure 17 convert from the temporal encoding to the others. They require a bias neuron, which provides a reference point to the spiking of the input neuron. The networks with leak are straightforward, where the input's timing with respect to the bias is easily converted into the proper output encoding.

The *no-leak* networks are again more complicated than the leak networks, because residual potential must be canceled rather than leaking away. It is interesting to note that the only cycles in the conversion networks are in the *no-leak* situations.

In Table 2, We tabulate the same metrics as in Table 1, for the conversions. Although we don't draw the networks here, the networks that convert from bins are the smallest and fastest, containing the least amount of activity. Spike and temporal encoding require larger networks with more activity and slower reusability.

8 CONCLUSIONS AND FUTURE WORK

We have presented networks and spike raster plots for 24 implementations of binary operations, and 24 conversions of one binary encoding to another. These networks will be useful as glue, preprocessing and post-processing networks for larger neuromorphic networks, such as those trained by backpropagation, STDP and genetic algorithms. Their most attractive feature is that they implement their functionalities on a neuromorphic computing system, and do not require information to flow from a neuroprocessor to conventional hardware and back again, which is expensive from both a time and power perspective. They may also be used as building blocks for genetic training methods such as EONS [17].

Conversion	Leak					No Leak				
	N	S	A	O	R	N	S	A	O	R
$D \rightarrow B$	4	3	2.5	1	1	Same				
$D \rightarrow S$	3	2	3.0	1	1	Same				
$D \rightarrow T$	3	2	2.5	1	2	Same				
$B \rightarrow D$	3	1	0.5	1	1	Same				
$B \rightarrow S$	3	3	1.5	1	1	Same				
$B \rightarrow T$	3	2	1.0	1	2	Same				
$S \rightarrow D$	2	2	2.0	2	3	2	3	2.0	2	3
$S \rightarrow B$	4	6	3.0	3	3	4	7	3.0	3	3
$S \rightarrow T$	4	7	3.5	3	4	4	7	3.0	3	4
$T \rightarrow D$	3	2	2.5	1	2	3	4	2.5	1	2
$T \rightarrow B$	4	4	3.0	2	3	5	6	3.5	2	2
$T \rightarrow S$	4	4	4.0	2	3	4	5	3.5	2	3

Table 2: Metrics of interest for the conversion networks.

These networks may also be useful as stress-tests for neuromorphic computing systems, as they may be composed to create very large networks, whose calculations are easy to confirm. We plan to use them as comparison points for current neuroprocessors such as Loihi [7] and Caspian [14]. In fact, these networks should compose seamlessly with Intel's research simulator, Lava, for stress testing in both simulation and hardware.

For future work, we plan to extrapolate this research to numerical inputs and outputs, providing larger, richer networks that perform more complex calculations. Once again, the intent is not to replace machine learning approaches to RSNN training, but to augment them and allow for their composition.

We have written an interactive tool for animating all of these network, which is available from http://neuromorphic.eecs.utk.edu/, linked from this paper's web page.

ACKNOWLEDGMENTS

Research sponsored in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy. This research was also supported in part by an Air Force Research Laboratory Information Directorate grant (FA8750-19-1-0025).

REFERENCES

[1] J. B. Aimone, K. E. Hamilton, S. Mniszewski, L. Reeder, C. D. Schuman, and W. M. Severa. 2018. Non-neural network applications for spiking neuromorphic hardware. In *Proceedings of the Third International Workshop on Post Moore's Era Supercomputin*. 24–26.

[2] J. B. Aimone, Y. Ho, O. Parekh, C. A. Phillips, A. Pinar, W. Severa, and Y. Wang. 2020. Provable Neuromorphic Advantages for Computing Shortest Paths. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 497–499.

[3] J. B. Aimone, W. Severa, and C. M. Vineyard. 2019. Composing neural algorithms with Fugu. In *International Conference on Neuromorphic Computing Systems (ICONS)*. ACM, 1–8.

[4] J. D. Ambrose, A. Z. Foshie, M. E. Dean, J. S. Plank, G. S. Rose, J. P. Mitchell, C. D. Schuman, and G. Bruer. 2020. GRANT: Ground Roaming Autonomous Neuromorphic Targeter. In *IJCNN: The International Joint Conference on Neural Networks*.

[5] G. Bellec, F. Scherr, E. Hajek, D. Salaj, R. Legenstein, and W. Maas. 2021. Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets. arXiv:1901.09049.

[6] J. Cabessa. 2019. Turing complete neural computation based on synaptic plasticity. *PloS One* 14, 10 (2019), e0223451.

[7] M. Davies *et al.* 2018. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* 38, 1 (2018), 82–99. https://doi.org/10.1109/MM.2018.112130359

[8] J. Dupeyroux. 2021. A Toolbox for Neuromorphic Sensing in Robotics. arXiv:2103.02751v1.

[9] G. A. Fonseca Guerra and S. B. Furber. 2017. Using stochastic spiking neural networks on SpiNNaker to solve constraint satisfaction problems. *Frontiers in Neuroscience* 11 (2017), 714.

[10] B. Kay, P. Date, and C. Schuman. 2020. Neuromorphic Graph Algorithms: Extracting Longest Shortest Paths and Minimum Spanning Trees. In *NICE: Neuro-Inspired Computational Elements Workshop*.

[11] W. Maass. 1996. Lower bounds for the computational power of networks of spiking neurons. *Neural Computation* 8, 1 (1996), 1–40.

[12] W. Maass, T. Natschläger, and H. Markram. 2002. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation* 14, 11 (2002), 2531–2560. https://www.mitpressjournals.org/doi/10.1162/089976602760407955

[13] T. Masquelier, R. Guyonneau, and S. J. Thorpe. 2008. Spike Timing Dependent Plasticity Finds the Start of Repeating Patterns in Continuous Spike Trains. *PLoS ONE* 3, 1 (2008). https://doi.org/10.1371/journal.pone.0001377

[14] J. P. Mitchell, C. D. Schuman, R. M. Patton, and T. E. Potok. 2020. Caspian: A Neuromorphic Development Platform. In *NICE: Neuro-Inspired Computational Elements Workshop*. ACM. https://doi.org/10.1145/3381755.3381764

[15] S. M. Mniszewski. 2019. Graph Partitioning as Quadratic Unconstrained Binary Optimization (QUBO) on Spiking Neuromorphic Hardware. In *International Conference on Neuromorphic Computing Systems (ICONS)*. ACM, 1–5.

[16] J. V. Monaco and M. M. Vindiola. 2017. Integer Factorization with a Neuromorphic Sieve. *CoRR* abs/1703.03768 (2017). http://arxiv.org/abs/1703.03768

[17] C. D. Schuman, J. P. Mitchell, R. M. Patton, T. E. Potok, and J. S. Plank. 2020. Evolutionary Optimization for Neuromorphic Systems. In *NICE: Neuro-Inspired Computational Elements Workshop*.

[18] C. D. Schuman, J. S. Plank, G. Bruer, and J. Anantharaj. 2019. Non-Traditional Input Encoding Schemes for Spiking Neuromorphic Systems. In *IJCNN: The International Joint Conference on Neural Networks*. Budapest, 1–10. https://doi.org/10.1109/IJCNN.2019.8852139

[19] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank. 2017. A Survey of Neuromorphic Computing and Neural Networks in Hardware. arXiv:1705.06963. https://arxiv.org/abs/1705.06963

[20] W. Severa, R. Lehoucq, O. Parekh, and J. B. Aimone. 2018. Spiking neural algorithms for markov process random walk. In *IJCNN: The International Joint Conference on Neural Networks*. IEEE, 1–8.

[21] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone. 2019. Training Deep Neural Networks for Binary Communication with the Whetstone Method. *Nature Machine Intelligence* 1 (January 2019), 86–94. https://doi.org/10.1038/s42256-018-0015-y

[22] S. B. Shrestha and G. Orchard. 2018. SLAYER: Spike Layer Error Reassignment in Time. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 1412–1421. http://papers.nips.cc/paper/7415-slayer-spike-layer-error-reassignment-in-time.pdf

[23] J. D. Smith, W. Severa, A. J. Hill, L. Reeder, B. Franke, R. B. Lehoucq, O. D. Parekh, and J. B. Aimone. 2020. Solving a steady-state PDE using spiking networks and neuromorphic hardware. In *International Conference on Neuromorphic Computing Systems (ICONS)*. ACM, 1–8.

[24] S. J. Verzi, R. Rothganger, O. D. Parekh, T. T. Quach, N. E. Miner, C. M. Vineyard, C. D. James, and J. B. Aimone. 2018. Computing with Spikes: The Advantage of Fine-Grained Timing. *Neural Computation* 30, 10 (October 2018), 2660–2690. https://www.mitpressjournals.org/doi/abs/10.1162/neco_a_01113

[25] C. Yakopcic, N. Rahman, T. Atahary, T. M. Taha, and S. Douglass. 2020. Solving constraint satisfaction problems using the Loihi spiking neuromorphic processor. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1079–1084.