A Spike-Based Long Short-Term Memory on a Neurosynaptic Processor

Amar Shrestha[1], Khadeer Ahmed[1], Yanzhi Wang[1], David P. Widemann[2], Adam T. Moody[2], Brian C. Van Essen[2], Qinru Qiu[1]
[1]*Department of Electrical Engineering and Computer Science, Syracuse University, NY 13244, USA*
[2]*Lawrence Livermore National Laboratory, Livermore, CA 94551 USA*
[1]{ amshrest , khahmed, ywang393, qiqiu} @syr.edu [2]{ widemann1, moody20, vanessen1} @llnl.gov

***Abstract*---Low-power brain-inspired hardware systems have gained significant traction in recent years. They offer high energy efficiency and massive parallelism due to the distributed and asynchronous nature of neural computation through low-energy spikes. One such platform is the IBM TrueNorth Neurosynaptic System. Recently TrueNorth compatible representation learning algorithms have emerged, achieving close to state-of-the-art performance in various datasets. An exception is its application in temporal sequence processing models such as recurrent neural networks (RNNs), which is still at the proof of concept level. This is partly due to the hardware constraints in connectivity and synaptic weight resolution, and the inherent difficulty in capturing temporal dynamics of an RNN using spiking neurons. This work presents a design flow that overcomes the aforementioned difficulties and maps a special case of recurrent networks called Long Short-Term Memory (LSTM) onto a spike-based platform. The framework is built on top of various approximation techniques, weight and activation discretization, spiking neuron sub-circuits that implements the complex gating mechanisms and a store-and-release technique to enable neuron synchronization and faithful storage. While many of the techniques can be applied to map LSTM to any SNN simulator/emulator, here we demonstrate this approach on the TrueNorth chip adhering to its constraints. Two benchmark LSTM applications, parity check and Extended Reber Grammar, are evaluated and their accuracy, energy and speed tradeoffs are analyzed.**

Keywords--Spiking Neural Networks, Recurrent Neural Networks, Long Short-Term Memory, Neuromorphic Hardware

I. INTRODUCTION

Inspired by neurons, building blocks of our brain, and their connections, various versions of artificial neural networks have been designed and achieved high performance in representation learning tasks such as image classification and pattern recognition. However, these feedforward neural networks are not capable of retaining temporal dependencies in a sequence, and are not suitable to learn temporal patterns from time-series where the information at the current time step is dependent on past steps. While recurrent neural networks (RNNs) address this issue with feedback connections, it is difficult to learn long temporal dependencies using vanilla RNNs [1]. Long Short-Term Memory (LSTM) improves RNN with a complex gated mechanism, which allows it to forget, remember and output information [2]. Its ability of learning long-term dependencies makes it a prominent and successful model for time-series processing.

The advents of energy-efficient large-scale neuromorphic hardware enabled low power implementation of large-scale neural networks for real-time applications. One of the examples is IBM's Neurosynaptic Processor, "TrueNorth". Operating in the spiking-domain, TrueNorth has achieved close to state-of-the-art results in various pattern recognition tasks [3] with very high energy efficiency. Converting pre-trained network to a SNN has also produced good results in pattern recognition [4] on platforms other than TrueNorth. However, almost all of these applications aim at non-recurrent networks, such as convolutional neural networks. Due to the hardware constraints in connectivity and synaptic weight precision, and the inherent difficulty in capturing temporal dynamics of an RNN using spiking neurons, implementing recurrent neural networks (RNNs) for temporal sequence processing in spike-domain is still at the proof of concept level [5].

In this work, we present a design flow that overcomes the aforementioned difficulties and maps LSTM, a special case of RNN, onto a spike-based platform, and implement them using the TrueNorth processor. We validate the implementation using two benchmark LSTM models. The framework is built on top of various approximation techniques, including weight and activation discretization, spiking neuron sub-circuits that implements the complex gating mechanisms and a store-and-release technique that enables neuron synchronization and faithful storage.

To the best of our knowledge, there has been no publication on implementing LSTM in spike domain. The following summarizes our contributions:

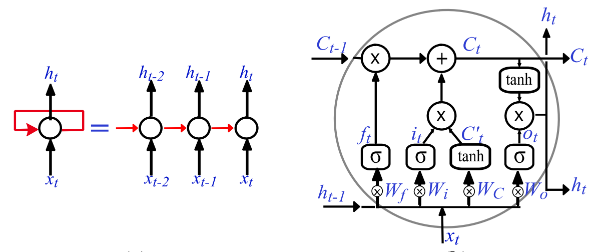
1. We developed a modular approach to convert a standard LSTM to a Spiked-based LSTM. The modular approach allowed for incremental mapping onto the TrueNorth chip.
2. To have a faithful representation of inputs, outputs and internal activation of an LSTM in spike-domain, encoding heuristics are adopted, which maintain spike representation consistency throughout the network.
3. Novel neuron circuit design is proposed to approximate the sigmoid and hyperbolic tangent functions. The relationship between stored membrane potential, the random firing threshold and the firing rate is analyzed.
4. To synchronize the gated modules and achieve recurrent processing in the Spike-based LSTM, we developed a store-and-release mechanism using locally generated and globally consistent store and release clock spikes.

II. BACKGROUND

A. LSTM

The main feature of a recurrent neural network is that it can learn sequential information by considering the information from previous time steps. The loop, as shown in Figure 1(a), allows information to be passed from one step of the network to the next thus allowing the information to persist.

The length of sequence or how far a vanilla RNN can remember is hindered by vanishing or exploding gradients [1] since



(a) **(b)**

$$f_t = \sigma(W_f . [h_{t-1}, x_t] + b_f) \qquad i_t = \sigma(W_i . [h_{t-1}, x_t] + b_i)$$
$$o_t = \sigma(W_o . [h_{t-1}, x_t] + b_o) \qquad C'_t = tanh(W_C . [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * C'_t \qquad h_t = o_f * \tanh(C_t)$$

(c)

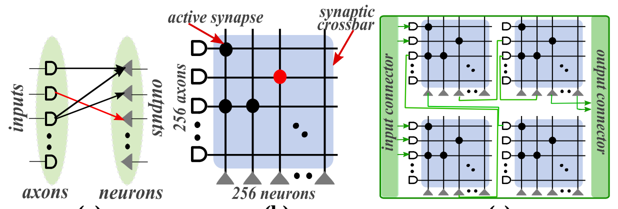
Figure 1(a)Unrolled RNN (b)Standard LSTM (c) LSTM equations

backpropagation-through-time results in an unrolled network, which can be very deep. To overcome this, LSTM utilizes a special structure of gates to selectively allow information to persist in the cell state. Its structure provides the ability to remove or add information to the cell state, carefully regulated by gates. Gates are a way to optionally let information through. They are composed out of a sigmoid and a pointwise multiplication operation. The input gate, forget gate and the output gate allowed for adding, removing and outputting information to or from the cell state. This makes LSTM successful in tasks like language modeling, machine translation, speech recognition, video to text etc.

There are many variations of LSTM such as Gated Recurrent Units (GRU) [6], Peephole LSTM [7], etc. In this work, we aim at a standard LSTM model [2], which is shown in Figure 1(b). A single standard LSTM has a forget, input and output gate which are sigmoid activations, and hyperbolic tangent activations to squash incoming inputs and outgoing output. Its output h_t is generated based on the equations shown in Figure 1(c), where f_t, i_t and o_t are the forget, input and output gate vectors, c_t and h_t are cell state and output vectors, and W_f, W_i, W_o and W_C are trained weight matrices.

B. *TrueNorth Architecture*

The TrueNorth neurosynaptic processor is inspired by the parallel architecture of biological neural systems. It is highly efficient, scalable and flexible. The TrueNorth processor consists of 4096 cores [8] each with 256 neurons and 256 axons connected via 256x256 directed synaptic connections, thus providing 1 million programmable neurons and 268 million configurable synapses. The weight of the corresponding synapse in the crossbar is selected from 4 possible integers determined by the axon type at each neuron.



(a) **(b)** **(c)**

Figure 2(a)Structural view (b)Functional view of a core (c) corelet

TrueNorth uses an efficient event-driven architecture where, address event representation (AER) is adopted for spike representation and communication between neurons. These spike events are sparse in time and active power is proportional to firing activity thus making it highly efficient and low power. Normally, the system operates in 1 ms timesteps called ticks within which membrane potential is processed and spike events routed asynchronously inside the chip. A spike generated by a neuron can target any single axon on the chip. Figure 2(a) shows a structural view of a TrueNorth core with axons as input and neurons as outputs and synapses linking them. This representation is similar to a traditional neural network. Figure 2(b) shows a functional view of core as a crossbar where horizontal lines are axons, cross points are individually programmable synapses, vertical lines are neuron inputs, and triangles are neurons. Spikes flow from axons via active synapses to neurons.

The synaptic connections and their weights between axons and neurons are captured by a crossbar matrix at an abstract level along with connections from neuron to a single axon across cores. This abstraction is called a Corelet [9], which represents a network on the TrueNorth cores by encapsulating all details except external inputs and outputs as shown in Figure 2(c). The creating, composing and decomposing of corelets is done in an object-oriented Corelet Language in the programming paradigm for TrueNorth called Corelet Programming Environment (CPE). Programming TrueNorth includes creating corelets with specific neuron behaviors, synaptic connections, weights and delays to achieve the desired functionalities. Multiple corelets can be combined through their input and output connectors.

III. PROPOSED IMPLEMENTATION

Event driven neuron operation and asynchronous inter-core communication are representative features of many hardware implementations of Spiking Neural Networks (SNNs) [8] [10] [11] including the TrueNorth processor. It reduces the hardware active power, however, also imposes a fundamental challenge to realize the LSTM. As shown in Figure 1(a), the proper function of an LSTM relies on the synchronization of neuron inputs. For example, the output vector o_{t-1} in time step *t*-1 must concatenate with the input vector x_t in time step *t* to calculate the new output vector. On an event-driven hardware platform, such as TrueNorth, special neural circuits must be designed to enable such synchronization. Other challenges of implementing the LSTM on a neurosynaptic processor in the spike-domain include a lack of low precision algorithms which can achieve results typically observed when using floating point precision based non-linear activation functions such as sigmoid and tanh, and the difficulty in representing numerical values in spike domain.

In this work, we address the aforementioned challenges and present some key techniques that facilitate Spike-based LSTM and its mapping onto the TrueNorth chip. We start with approximations made to the LSTM and our constrain-then-train-then-approximate process, which minimizes the approximation errors. In the next section, we discuss how values are represented using spikes along with the external and internal encoding schemes of the Spike-based LSTM. Then we describe the constituent modules of LSTM on TrueNorth and how to maintain the temporal relation of these modules' activities. Finally, we present the mapping algorithm.

A. *"Constrain-Then-Train-Then-Approximate"*

Several approximations on the LSTM are adopted during the mapping to TrueNorth. We consider these approximations during training to minimize potential errors. We call this process "constrain-then-train-then-approximate."

The synaptic weights in TrueNorth have limited precision and coarse granularity. They are represented using 9-bit signed binary data, and there can be at most 4 different weights for all synapses connecting to the same axon. Although binary and ternary weights have been used to produce close to state-of-the-art results for feed-forward network architectures [12] [13], both our preliminary work and the existing research [14] show that it is difficult to train LSTMs with binary and ternary weights. Power2-ternarization, which rounds the integer part and fractional part of the weight separately [14], gives good training results. However, it only provides an efficient way to discretize the weights. High precision data is still needed for the LSTM to work properly. Instead of training the network using ternary weights, in this work, we approximate the weights of a regularly trained LSTM via scaling and rounding.

Due to the hardware constraints, the non-linear activation functions such as tanh and sigmoid gates are also approximated using piece-wise linear functions. Their implementation details will be discussed in Section 3.4.

To minimize the potential errors, we constrain the network during the training to reflect those approximations. Firstly, the LSTMs are trained using constrained weights (-1 to 1 or -2 to 2) such that these weights can be scaled to a hardware supported range. Secondly, we replace the gates (tanh and sigmoid) with their piece-wise linear counterparts (hard tanh and hard sigmoid), which have steeper slopes as shown in Figure 7(a) and (b) and in Eqn. (2) and (4). The weights of the trained network are floating point values thus they are scaled and rounded off to the required precision and range while converting to spike domain. This is done by approximation.

B. *Temporal Behavior of Neurons in LSTM*

The complex gating mechanism of the LSTM requires synchronization of inputs, gate outputs and the cell state feedback. Synchronization is also necessary to maintain the temporal dynamics of the recurrence of the LSTM output in the network. Representing this level of synchronization using spiking neurons is inherently difficult given their asynchronous nature. We overcome this limitation using a *store-and-release* mechanism. It is built atop a class of special neural circuits, in which neurons operate in two modes, *store* and *release*. During the store mode, the neurons gate their outputs, receives input spikes and at the same time accumulate their membrane potential. During the release mode, the neurons issue output spikes at an average rate proportional to its membrane potential either stochastically or in a burst. Two internal clock signals controls when a neuron enters or exits the store/release modes through the application of highly negative/positive potential respectively. How to configure and connect asynchronous neurons to form such synchronous neural circuits will be discussed in Section 3.4.

Using the store-and-release mechanism, the LSTM works in two phases, *processing phase* and *I/O phase*. The entire LSTM network is divided into three partitions as shown in Figure 3. (a) and (b). They are referred to as input, processing and output partitions and color coded using blue, red and green respectively. Except the input partition, all inputs of the processing and output partitions are buffered using store-and-release neurons. The inputs to the processing partition stores during the I/O phase and releases in the processing phase, while the inputs to the output partition stores in the processing phase and releases in the output phase. For the input partition, one of its inputs h_{t-1} is released only in the I/O phase, and by careful control we can also make sure that the external inputs, x_t , are issued only during the I/O phase, therefore, the input partition is active and releases output spikes only during the I/O phase. During the implementation, the store-and-release neurons will be merged into their subsequent function modules and be implemented as store-and-release tanh or store-and-release sigmoid, as we will present in Section 3.4. The only exception is the store-and-release neurons before the dot product, which will stay stand-alone.

How these three partitions operate alternatively is shown in Figure 3. (c). The duration of each phase is referred as phase length (PL). During the I/O phase, the input partition works on the matrix-vector multiplications to generate the operands for the forget, input and output gates. The results of the matrix-vector multiplications are buffered by the store-and-release neurons at the input of the processing partition. In the processing phase, these neurons release what they have stored and the processing partition generates the cell state (C_t) and partial output (o_t), which are buffered by the store-and-release neurons at the input of the output partition. During the next I/O phase, the C_t and o_t will be released and be used to calculate the h_t, which will be forwarded to the input partition to calculate the matrix-vector multiplications again.

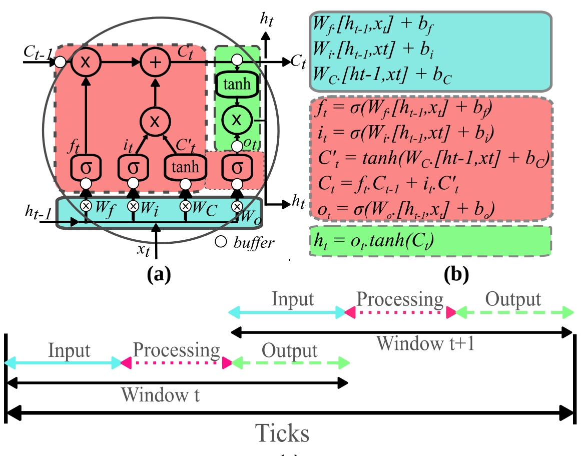


Figure 3. (a) LSTM color coded based on operation phase (b) LSTM equations color coded to represent operations in specific phases (c) 3 phases and partial pipelining

These phases are maintained through local clock (globally consistent) spikes which produce the store and release actions for all the store-and-release capable modules.

C. *Encoding and Spike Representation*

In an LSTM, numerical values (inputs, outputs and activations) have ranges in both negative and positive direction as the weights learnt can be negative and also the tanh function outputs values in the range -1 to 1. Since the spikes are binary (on-off), there is an inherent difficulty in representing both positive and negative values with a single channel of spikes. A simple solution is to constrain the values during training to a positive range by replacing tanh with ReLU. ReLUs have produced improvements for vanilla RNNs [15] due to its ability to stop vanishing. However, vanishing gradients is no longer a problem in LSTM due to its gating scheme. On the contrary, using unbounded activation functions like ReLU in an LSTM can cause it to diverge thus resulting in worse performance [16]. Therefore, we avoid replacing tanh with ReLUs and instead represent positive and negative values using a positive and negative channel of spikes respectively.

The inputs and outputs of an LSTM are rate-coded where the firing rate is determined by the phase length (PL) and the max value (mx) to be represented in that phase. If we scale up the trained weights with a scaling factor sf, the input and output should be scaled down by the same factor. Therefore, the number of spikes (nS) needed to represent value 1 can be calculated as:

$$nS = PL/(mx * sf).$$

To represent a numerical value Iv, the spike firing rate is set to $(Iv * nS)/PL$, and n spikes in a phase represent the value:

$$RPValue = \frac{\#spikes}{nS}.$$

The choice of mx and phase length determines the precision of values that can be represented by spikes, as each spike represents $1/nS$ in terms of numerical value.

All internal variables are rate-coded, except C_t. We found that the cell state C_t needs to be represented with higher precision, because any error on this variable will be accumulated due to the feedback path. The stochastic rate coding provides convenience in implementing multiplication as it requires only an AND function, however, it introduces not only rounding error but also random error due to stochastic sampling. Previous work shows that the spike burst code, where the numerical value is represented by the number of spikes that burst in a window, has much higher correlation with the numerical value to be represented [17]. Therefore, we encode C_t using spike burst code and use spike-burst neurons for the sum function.

D. *Spike-based LSTM Constituent Modules*

Figure 1(a) shows a general LSTM unit consisting of the sigmoid gates, hyperbolic tangent, dot products and sum. To implement a Spike-based LSTM (S-LSTM), we approximate these modules using spiking neurons. On TrueNorth, these modules are in the form of corelets, which will be further connected to form the full S-LSTM. The corelets will be mapped across cores based on the consideration of the fan-in and fan-out constraints of the hardware.

1) *Store-and-Release neurons*

Store and release mechanism is implemented using a neuron with a high negative threshold where it saturates. The store and release clocks are two inputs associated with large negative and positive weights respectively. A spike on the store clock pushes the membrane potential to the negative threshold and turns on the store mode, during which the neuron accumulates the input spikes and raises its membrane potential. The negative initial state guarantees that the raised membrane potential is still below the firing threshold, therefore, no output spikes are generated. A spike on the release clock pushes the membrane potential to 0 or higher if input spikes are received during the store model, and the neuron starts generating output spikes.

A problem with the above scheme is that the neuron cannot collect negative spikes at the beginning of the store mode as its membrane potential cannot go below the negative threshold. So, upon entering the store mode, we send a spike on the release clock, to pre-charge the membrane potential to an intermediate level between the negative threshold and 0 to allow collecting negative spikes. We refer to this spike signal as pre-charge clock. Figure 4 shows an example where the neuron enters the store mode through a negative potential of -250 and enters release mode through two positive potentials of +100 administered by the store (red), pre-charge (yellow) and release (blue) clock spikes respectively. Figure 5 shows all the clock signals in the S-LSTM.

At the beginning of the release phase, the neuron membrane potential (*AMP*) equals to the total number of net input spikes that it collected during the store mode. The $RPValue$ stored in the neuron can be calculated as $RPValue = AMP/nS$. To generate the output spikes, a random number is drawn in the range [0, *RThR*], where *RThR* is the firing threshold. If this number is less than the *AMP*, then an output spike is generated. During a phase *PL*, the expected number of spikes generated in this way is $PL \times AMP/RThR$. When we set *RThR* to *PL*, the number of the output spikes equals to the total number of net input spikes, and the neuron relays input to the output without any transformation. In the actual implementation, almost all store-and-release neurons are merged to its subsequent sigmoid and tanh gate. $RThR$ should be selected differently due to the squash and linear transformation of these functions. More details will be given in section III.D.3).

2) *Input Collection Module (IC Module)*

From the LSTM equations, we see that the weight matrices W for each gate are multiplied by the input vector x and the previous time-step's LSTM output vector h_{t-1} respectively along with the biases. To implement this matrix-vector multiplication, we develop this parameterized module. As shown in Figure 6(d), each input and output is represented using two channels to accommodate both positive and negative values. We use 4 axons with assigned weights 1,2,4 and 8 to approximately represent weights from -15 to +15. The two-channel weight mapping with 4 axons is capable of approximating 5-bit precision weights instead of 4-bit precision [18]. The absolute weight is assigned to the positive or negative output channel based on the resultant sign of the product of input and axon weight. For example, if the input is negative and the weight is positive, the resultant product is

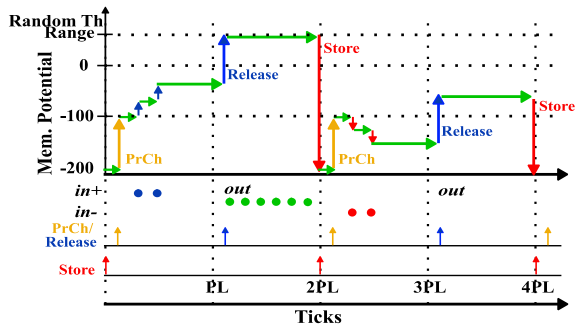


Figure 4. Store-and-release mechanism

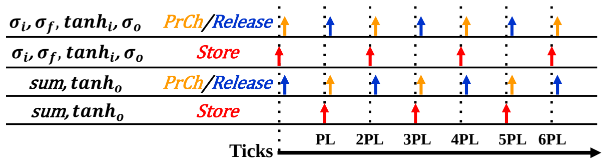


Figure 5. Store and release clock spikes for all gates

negative. Thus, the absolute weight is assigned to the negative output channel.

The above design results in positive and negative outputs in their respective channels. We use ReLU neurons for the output, which produces bursts of spikes equal to the accumulated membrane potential when threshold is 1. The two-channels (i.e. positive and negative) are used in every input of the sigmoid and tanh gates in the processing partition, which will then merge them to achieve the net results.

This module is also parameterized to accommodate matrices/vectors [W_i, W_h, X, Y] of various sizes and the mapping scales across multiple cores depending on the sizes of those matrices/vectors.

3) *Gate Modules*

Piece-wise linear functions are computationally efficient [19] due to minimal cut points and linear interpolation between those cut points. These cut points and linearity are more conducive than a smooth non-linearity to rate coding where the spiking rate determines the computed values. Spiking rates of these gate modules have definite max and min (0 and 1) and within this range the rate is linearly proportional to the number of input spikes or the accumulated membrane potential of a neuron. A steeper slope of the activation function reduces the linear range hence, limits the propagation of rounding errors.

LSTM uses sigmoid gates to allow the flow of input, cell state and output, and uses tanh to squeeze the inputs and outputs to a range. Given the constrain-then-train approach, we develop modules which produce a hard sigmoid and a hard tanh behavior with store and release capability, and use them during both training and recall. In IC module, the matrix-vector multiplication produces two separate channels (positive and negative). These spikes are collected in their respective gate modules to produce the net accumulated membrane potential (AMP) during the store mode.

During the release mode, spikes generated by the gate modules are rate-coded and the firing rates are linearly dependent on the ratio of total AMP and $RThR$.

For the sigmoid,

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

We approximate this as a piece-wise linear function with a steeper slope than used in [12] and TensorFlow [20] for a hard sigmoid.

$$\sigma(x) = \max(0, \min(1, x * 1 + 0.5)) \tag{2}$$

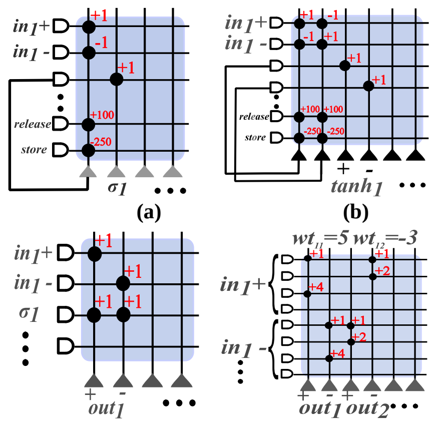


Figure 6. (a) Sigmoid module (b) Tanh module (c) Dot product module (d) IC module

Here, x is the $RPValue$ ($= AMP/nS$) accumulated from the input. $\sigma(x)$ has a range [0, 1], which is represented by the firing rate of its output spike. To represent $(x * 1 + 0.5)$, an offset of $nS/2$ is added to the AMP, so that the $RPValue$ becomes $(AMP + 0.5nS)/nS$. To ensure that the resultant firing rate saturates to 0 and 1 at $x = -0.5$ and $x = 0.5$ respectively, and be linearly proportional to the $RPValue$ for $-0.5 < x < 0.5$, we set $RThR = nS$.

Similarly, for the hyperbolic tangent,

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \qquad (3)$$

We approximate this as a piece-wise linear function with a steep slope. We choose slope=2 such that x saturates beyond $|0.5|$.

$$tanh(x) = \max(-1, \min(1, x * 2)) \qquad (4)$$

Again, x is the $RPValue$ ($= AMP/nS$) and $tanh(x)$ is the resultant firing rate with range $[-1,1]$. We approximate this using two output channels, for positive and negative x.

$$tanh(x) = \begin{cases} \max(-1, min(0, x * 2)), x < 0 \\ \max(0, \min(1, x * 2)), x \geq 0 \end{cases} \qquad (5)$$

And we choose $RThR = nS/2$ to reflect the scaling and get the behavior of a hard tanh in Eqn. (4).

4) *Dot Product Module*

As shown in Figure 6(c), the inputs of the dot product module are two-channeled inputs C_t, which is burst coded, and a rate-coded sigmoid. The function of the dot product is to help the gating functions, by allowing a certain percent of the information to flow through. Because the output of sigmoid function has numerical values between 0 and 1 and is stochastically rate-coded, we explore the stochastic nature of the input and perform the multiplication by stochastic computing. A simple logical AND of the two spike streams is used as the multiplication.

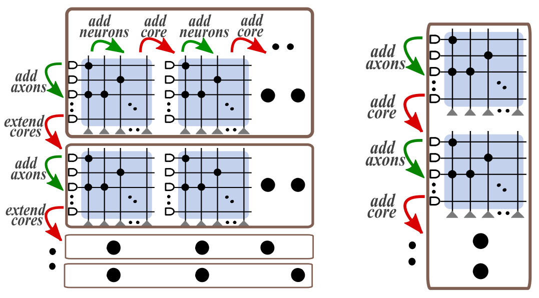


Figure 8(a) Algorithm 1 (b) Algorithm 2 in action

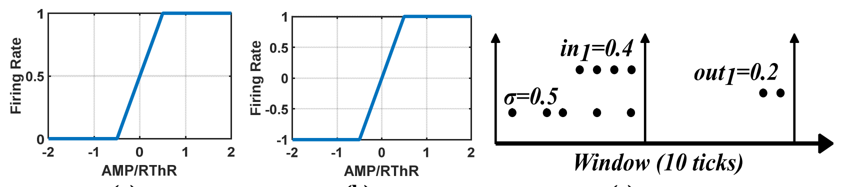


Figure 7. (a) Approximated hard sigmoid (b) Approximated hard tanh (c) Dot product through logical AND of spikes

An example is given in Figure 7(c). In a window of 10 ticks, there are 5 spikes from the sigmoid, which represents $\sigma = 0.5$, and 4 spikes in the in1, representing $in1 = 0.4$. The logical AND produces 2 spikes at the output representing $out1 = 0.2$. The logical AND operation can be easily realized using an integrate and fire neuron with threshold 2, by setting its input synapses weight and leak to be +1.

E. Mapping Algorithm

As a single TrueNorth core consists of 256 neurons and axons, there is a distinct fan-in and fan-out constraint. To deal with this constraint and freely map LSTM networks of arbitrary sizes, we develop an incremental mapping algorithm. The mapping of IC module and other modules vary slightly as the IC module's size is dependent on the number of LSTM units as well as the number of inputs whereas the sizes of other modules are only dependent on the number of LSTM units. Thus, IC modules are mapped and extended across two dimensions (axons and neurons) and the other modules are mapped and extended only across one dimension (axons). The mapping algorithms (Algorithm 1 and 2) are straightforward and, due to the incremental nature, cores are added only when the resources of the current core fills up and so on for each module as shown in Figure 8. This results in number of cores increasing in steps of the number of LSTM units and average neuron density (number of neurons used per core) increasing within those steps.

Algorithm 1. IC Module Mapping Algorithm

```
h = number of hidden units
x = number of inputs
function ExtendCores
    addCore
    numNeurons = 0
    for I = 1 to h
        add 2 neurons each for f, i, o, i_tanh
        numNeurons +=8
        if numNeurons>256
      addCore
      numNeurons = 0

call ExtendCores
numAxons = 0
for i = 1 to (x+h)
    add 8 axons (4 each for +ve and -ve)
    make synaptic connections as per 4 axon scheme
    numAxons += 8
    if numAxons > 256
        call ExtendCores
        numAxons = 0
```

Algorithm 2. Sigmoid, Tanh, Dot product Mapping Algorithm

```
h = number of hidden units
addCore
numAxons = 0
for i = 1 to h
    add a respective module with n axons
    numAxons += n
    if numAxons > 256
        addCore
        numAxons = 0
```

IV. EXPERIMENTS

In the following experiments, we compare the Spike-based LSTM mapped on TrueNorth against the standard LSTM implemented using Keras with TensorFlow backend. For all experiments, the network is comprised of one hidden layer of LSTM units and are trained with weights constrained to the range -2 to 2. Spike-based LSTM is set to mx=5 and results are noted for various phase lengths.

A. *Parity check / XOR problem*

Parity check of a bit stream is a classic problem that is difficult to solve with a standard feed-forward network, but simple to solve with an RNN. In this problem, we have a sequence of binary inputs, and determine at each input whether the number of 1's observed so far in the sequence is even or odd. Here we train LSTM with one hidden layer containing 2, 4 and 10 LSTM units on 9000 varying length binary sequences with maximum length of 20. Then it is tested with 1000 sequences on TrueNorth with varying phase lengths.

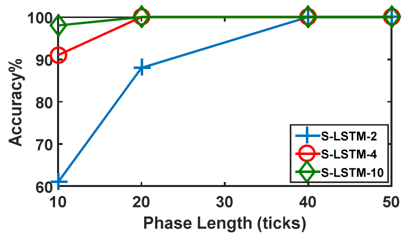


Figure 9. Accuracy vs Phase Length for Spike-based LSTM (Keras/Tensorflow counterparts have 100% for all cases)

In Figure 9, we can see that the performance improves with the phase length as this increases the precision of activations. The accuracy also increases when the number of LSTM units is increased.

B. *Embedded Reber Grammar*

Embedded Reber grammar (ERG) is a popular RNN benchmark [2] used by many authors and is useful for training sequences with short time lags. Figure 10 (a) shows a Reber Grammar graph which is extended to an Embedded Reber Grammar in Figure 10 (b). An ERG sequence starts from the leftmost node 'B' of the ERG graph, and sequentially generates a finite number of symbols by following edges until the rightmost node 'E' is reached. At some nodes, there can be two possible paths. This choice is made randomly.

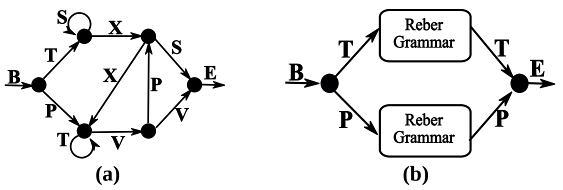


Figure 10. (a) Reber Grammar (b) Embedded Reber Grammar

Input and target patterns are represented by 7-dimensional binary vectors, representing one symbol each, in the training set. And the task is to read the symbols one at a time, and to continually predict the next possible symbol(s). Input vectors have exactly one nonzero component but the target vector could have one or two nonzero components representing one or two possible paths. The prediction is considered correct if it predicts either one or both possible symbols. Here we train LSTM networks with one hidden layer containing 10, 30 and 50 LSTM units on 5000 ERG sequences with maximum sequence length 36. We tested on 500 ERG sequences on TrueNorth with varying phase lengths.

Again, the accuracy is directly correlated to the phase length such that as the accuracy drastically improves for precision higher than $1/10$ ($1\ spike = mx/PL\ value$). The trend is noticeable in all 3 network sizes as shown in Figure 11.

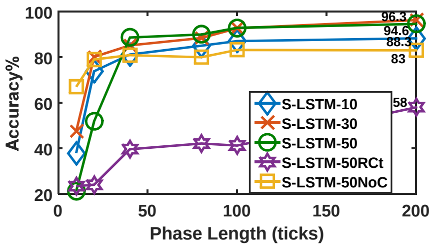


Figure 11. Accuracy vs Phase Length (PL) for Spike-based LSTM (Keras/Tensorflow counterparts have 100% for all cases). RCt=Rated coded Ct, NoC = No Constraints

Table 1. Power and performance on different platforms

Network	Devices	Time/ Sample (ms)	Active Power (W)	Energy/ Sample (mJ)
50-LSTM	NVIDIA Tegra X1	1.928	2.45	4.72
	NVIDIA Titan X	0.5	46.5	23.3
50-LSTM 200-PL	IBM TrueNorth	400	0.00014	0.056
	*IBM TrueNorth	114	0.00025	0.0285

*running at faster than real time (3.5x faster)

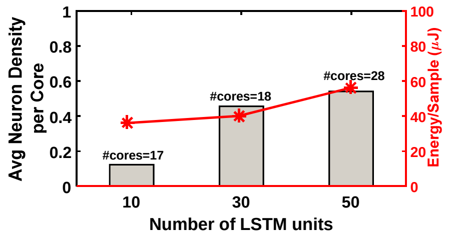


Figure 12. Neuron density, number of cores and energy/sample for the 3 S-LSTM networks

The network using rate-coded internal cell state (S-LSTM-50RC$_t$) instead of burst-coded one (S-LSTM-50) performs significantly worse. This, as mentioned in section C, is due to the accumulation of rounding error and additionally the sampling error while the spikes move from one buffer to another and then feeds back.

When trained without any constraints, the range of learned weights varies. If that range is wide, it is hard to find a scaling factor that raises the smaller values to hardware supported range for less rounding error without making the larger values to overflow. So, we simply set the scaling factor sf to be 1. Compared to the networks trained with constraints, the rounding error of the unconstrained network is higher. However, setting $sf = 1$ leads to higher nS and better precision (i.e. lower $1/nS$) than setting $sf > 1$. Therefore, the unconstrained network (S-LSTM-50NoC) produces better accuracy than the constrained network (S-LSTM-50) at lower PL, because it allows higher data precision. When the PL is high, the large window size already ensures reasonable data precision, so the constrained network performs better than the unconstrained version.

Table 1 shows the results of 50-unit LSTM network and the respective TrueNorth implementations with 200 (high accuracy) phase lengths. It shows that although time to process a sample is much less for NVIDIA Tegra X1 (20nm technology) and NVIDIA Titan X (16nm technology), the TrueNorth networks (running at normal operating frequency 1 kHz) is more energy efficient. It consumes only 56 μJ at 0.8V for 200 PL per sample making it up to 84x and 416x energy efficient than Tegra X1 and Titan X respectively. At faster than real time operation (3.5 kHz operating frequency) at the same voltage level of 0.8V, TrueNorth performs even better with 165x and 817x higher energy efficiency compared to Tegra X1 and Titan X.

The actual power consumption only for the resources utilized on TrueNorth chip is computed by measuring chip idle power which is leakage power P_{leak} and total power P_{total} with the network running [21]. Active power is $P_{active} = P_{total} - P_{leak}$. The scaled leakage power for the cores utilized is $P_{leak} = P_{leak}$*#cores/4096. Therefore, the total power consumed for the utilized resources is $P_{total\_s} = P_{active} + P_{leak\_s}$. For a given network the active power is directly proportional to the spiking activity however, the same network can be designed to be mapped utilizing different number of cores. To ensure minimum power consumption it is critical to pack as many neurons possible on to each core for minimizing P_{total_s} as, P_{leak_s} is the only free variable. Figure 12. shows the results of neuron packing density achieved by the proposed modular design of the Spike-based LSTM and the incremental mapping algorithms. We see that the neuron density is increasing as the number of LSTM unit increases for larger networks. It also shows the energy/sample for 50-S-LSTM, 200PL, 0.8V implementation.

Table 1 also shows the TrueNorth implementation is slower than the GPU implementations. In practical deployment, this factor of time delay is unsuitable and it points to a large performance/energy tradeoff. This tradeoff is drastic mainly due to the inability to reliably train LSTM networks with very limited precision. Here, we chose 200 PL setup for comparison to ensure reasonable data precision when we use $sf > 1$ to get reasonably high accuracy. If we can deploy an LSTM network trained with very limited precision, we can use $sf = 1$ without reducing the accuracy. And with that we can choose a much lower PL to get tolerable or even close to the GPU implementation delays with the same or even lower energy footprint as of now. Thus, minimizing the performance/energy tradeoff significantly.

V. CONCLUSION

The paper presents a Spike-based implementation of LSTM on the IBM TrueNorth Neurosynaptic Processor. A standard LSTM is divided into modules and separately approximated using spiking neurons. On TrueNorth, modules are in the form of corelets which are then combined, connected and mapped to form Spike-based LSTM networks and synchronized using a store-and-release mechanism. These networks are tested on two RNN benchmarks with promising accuracy results and high power efficiency.

REFERENCES

[1] Y. Bengio, P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks,* vol. 5, pp. 157-166, 1994.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735-1780, 1997.

[3] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch and others, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences,* p. 201604850, 2016.

[4] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *Neural Networks (IJCNN), 2015 International Joint Conference on*, 2015.

[5] P. U. Diehl, G. Zarrella, A. Cassidy, B. U. Pedroni and E. Neftci, "Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware," in *Rebooting Computing (ICRC), IEEE International Conference on*, 2016.

[6] J. Chung, C. Gulcehre, K. Cho and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555,* 2014.

[7] F. A. Gers, N. N. Schraudolph and J. Schmidhuber, "Learning precise timing with LSTM recurrent networks," *Journal of machine learning research,* vol. 3, pp. 115-143, 2002.

[8] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura and others, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science,* vol. 345, pp. 668-673, 2014.

[9] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza and others, "Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*, 2013.

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

[11] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras and S. B. Furber, "SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor," in *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, 2008.

[12] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830,* 2016.

[13] F. Li, B. Zhang and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711,* 2016.

[14] J. Ott, Z. Lin, Y. Zhang, S.-C. Liu and Y. Bengio, "Recurrent neural networks with limited numerical precision," *arXiv preprint arXiv:1608.06902,* 2016.

[15] S. S. Talathi and A. Vartak, "Improving performance of recurrent neural network with relu nonlinearity," *arXiv preprint arXiv:1511.03771,* 2015.

[16] T. M. Breuel, "Benchmarking of LSTM networks," *arXiv preprint arXiv:1508.02774,* 2015.

[17] Q. Chen and Q. Qiu, "Real-time Anomaly Detection for Streaming Data using Burst Code on a Neurosynaptic Processor," in *Proc. Conf. Design, Autom. Test Eur.(DATE)*, 2017.

[18] P. U. Diehl, B. U. Pedroni, A. Cassidy, P. Merolla, E. Neftci and G. Zarrella, "Truehappiness: Neuromorphic emotion recognition on truenorth," in *Neural Networks (IJCNN), 2016 International Joint Conference on*, 2016.

[19] A. Laudani, G. M. Lozito, F. R. Fulginei and A. Salvini, "On training efficiency and computational costs of a feed forward neural network: a review," *Computational intelligence and neuroscience,* vol. 2015, p. 83, 2015.

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin and others, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467,* 2016.

[21] A. S. Cassidy, R. Alvarez-Icaza, F. Akopyan, J. Sawada, J. V. Arthur, P. A. Merolla, P. Datta, M. G. Tallada, B. Taba, A. Andreopoulos and others, "Real-time scalable cortical computing at 46 giga-synaptic OPS/watt with," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2014.

[22] R. Preissl, T. M. Wong, P. Datta, M. Flickner, R. Singh, S. K. Esser, W. P. Risk, H. D. Simon and D. S. Modha, "Compass: A scalable simulator for an architecture for cognitive computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.