# Stochastic Computing Architectures for Lightweight LSTM Neural Networks

Roshwin Sengupta and Ilia Polian
*Institute of Computer Architecture and Computer Engineering*
*University of Stuttgart*
Stuttgart, Germany
{roshwin.sengupta, ilia.polian}@iti.uni-stuttgart.de

John P. Hayes
*Computer Engineering Laboratory*
*University of Michigan*
Ann Arbor, Michigan, USA
jhayes@umich.edu

***Abstract*—For emerging edge and near-sensor systems to perform hard classification tasks locally, they must avoid costly communication with the cloud. This requires the use of compact classifiers such as recurrent neural networks of the long short term memory (LSTM) type, as well as a low-area hardware technology such as stochastic computing (SC). We study the benefits and costs of applying SC to LSTM design. We consider a design space spanned by fully binary (non-stochastic), fully stochastic, and several hybrid (mixed) LSTM architectures, and design and simulate examples of each. Using standard classification benchmarks, we show that area and power can be reduced up to 47% and 86% respectively with little or no impact on classification accuracy. We demonstrate that fully stochastic LSTMs can deliver acceptable accuracy despite accumulated errors. Our results also suggest that ReLU is preferable to tanh as an activation function in stochastic LSTMs**

***Keywords—LSTM, recurrent neural nets, stochastic computing***

## I. Introduction

Emerging technologies will likely play a decisive role in making AI techniques, compact neural networks (NNs) such as LSTMs, practical for resource-constrained hardware systems. Among the many ways to implement NNs is *stochastic computing* (SC) [1]. SC computes with randomized bit-streams and can drastically reduce an NN's area and power needs while maintaining acceptable performance levels and offering other benefits such as error tolerance. Today, most NNs are implemented in the cloud, but lightweight hardware NNs, including SC-based ones, can enable near-sensor computing where key computations take place locally. This is attractive since communication links to the cloud and their associated drawbacks are greatly reduced. The suitability of SC for NNs has long been noted [2], but recent advances in SC have led to a plethora of new SC-based NN designs [3]. While most work focuses on SC implementation of convolutional NNs [4], recurrent NNs (RNNs) such as long short term memory NNs (LSTMs) [5] have attracted less attention, despite their advantages in area and power [6], [7]. One concern when using SC is their low accuracy [8]. Therefore, when considering SC for LSTMs, one must carefully balance gains in area and power consumption against potential losses in classification accuracy.

We propose several SC-based architectures for individual LSTM cells and NNs (Fig. 1). We consider both fully stochastic SC realizations and hybrid architectures where some submodules are stochastic, and some are binary. The SC-based LSTM is explored in-depth, using the full binary-based architectures of two representative networks TIMIT and JV from [6] and further, more complex networks MNIST, IMDB, TSC (Sec. IV.) as benchmarks for evaluation. We show that significant area reductions of 30-47% and power reductions up to 86 % can be achieved while keeping the classification accuracy at acceptable levels. Thus, SC offers attractive newpoints in the design space that trade cost for accuracy. We also study the impact of the LSTM's activation functions on the architectures considered. We find that ReLU is advantageous for SC-based modules, whereas in binary modules tanh performs better.

This work was supported in part by Deutsche Forschungsgemeinschaft (DFG), project number PO 1220/12-1. John P. Hayes was supported by the U.S. National Science Foundation under Grant CCF-2006704.

## II. Background

### A. Stochastic Computing

Stochastic computing (SC) [1] is an attractive technology for compact, error-tolerant, and low-power implementations of complex arithmetic functions. It has been used successfully in a variety of applications, such as low-density parity-check (LDPC) decoding [9], image processing [10], digital filter design [11], as well as NNs [4][6][7].

In SC, the basic number representation is a sequence of $n$ bits called a stochastic number (SN) whose value is determined by the frequency of 1s present. A unipolar SN has the value $n_1/n$ in the interval [0,1], where $n_1$ is the number of 1s in the SN. In bipolar SNs, the number range is extended to the interval [-1,1] by assigning the value $(n_1 - n_0)/n$ to the SN, where $n_0$ is the number of 0s it contains. For example, the SN $X_1$ = 10101101 has a value 5/8 in the unipolar representation, while in bipolar it has the value 2/8. Besides value range, another important difference between the two formats is *precision*, i.e., the smallest representable non-zero value. The precision of a unipolar SN is $1/n$ and in the bipolar format it is $2/n$. The precision reflects the contribution of each individual bit to the overall SN value. The order of 1s and 0s in an SN does not matter; so many SNs have the same numerical value. For example, $X_2$ = 11001110 is equivalent to $X_1$ = 10101101 above. These properties lead to the high error tolerance of SC; a few erroneous bits have a small impact on an SN's numerical value.

The basic SC operations, multiplication, and addition are shown in Fig. 2. The unipolar multiplication of two input SNs $a$ and $b$ is performed by logical AND gates. XNOR gates are used to multiply bipolar SNs. As the allowed range of unipolar SN is [0, 1], the normal add operation $a + b$ is unsuitable because the sum falls into [0, 2]. To ensure that the sum lies in [0, 1] the scaled addition $(a + b)/2$ is used in SC. A MUX performs scaled addition for both SN formats.

Randomness plays a central role in SC. SNs are produced by a stochastic number generator (SNG) which is commonly built around a pseudo-random number source like an LFSR. Given a binary number $B \in [0,1]$, the SNG outputs an $n$-bit unipolar SN with an expected value $B$. To generate a bipolar SN with value $B_b$ the binary input is set to $B = (B_b + 1)/2$. To

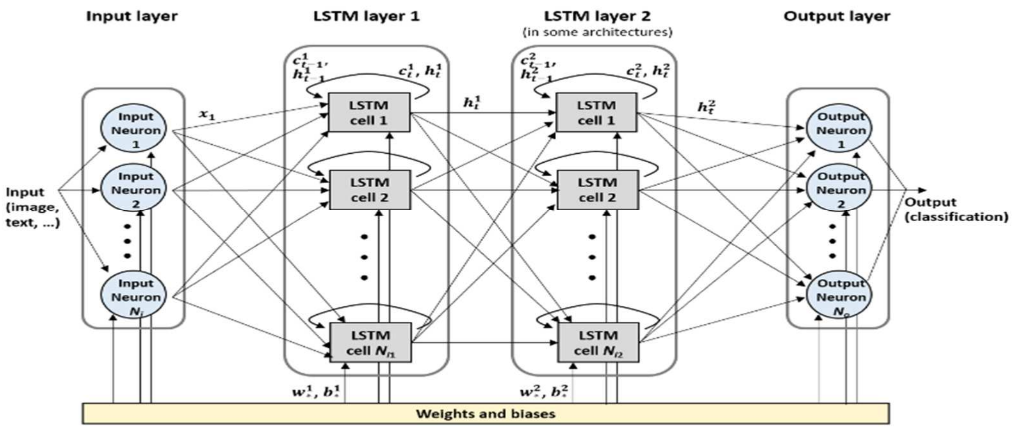

**Figure 1***:* Generic LSTM network. The symbols illustrated for LSTM cell 1 in the LSTM layers match Fig. 3.

convert an SN to the binary domain, a simple counter is used which counts in binary the number of 1s in the SN.

## *B. Long Short Term Memory (LSTM) networks*

LSTM [5] networks were developed for complex sequential data processing. They belong to the class of *recurrent neural networks* (RNNs) that, in contrast to traditional feed-forward NNs, contain feedback loops to capture time-dependent relationships between input/output sequences. Despite their successful use in speech recognition [12] and language translation [13], basic RNNs suffer from *vanishing* or *exploding gradients* problems [14]. The former occurs when changes in the early layers of the network fail to cause perceptible changes in the output. The latter problem refers to the scenario where insignificant changes in the early layers are amplified too much when propagated to the output layer.

An LSTM network is organized into at least one layer of *LSTM cells*, in addition to an input and an output layer (see Fig. 1). Each such cell has the structure shown in Fig. 3. An LSTM cell maintains a *cell state c* and a *hidden state h*, and its behavior is sequential. In step *t*, the cell updates its cell state $c_t$, and its hidden state $h_t$ based on its input $x_t$, weights and biases learned, and states $c_{t-1}$ and $h_{t-1}$; the states of all LSTM cells together serve as the network's memory. Conceptually, LSTM overcomes the vanishing gradient problem by explicitly specifying information, such as very small gradient values, that will be added to or removed from its state by each cell. To implement these features, an LSTM cell includes a *forget gate* that determines what past information is worth keeping, an *input gate* that decides what new information is relevant to add, and an *output gate* that calculates the current hidden state, as depicted in Fig. 3.

## III. Stochastic LSTM Architectures

The proposed SC architectures are based on comparing stochastic and conventional binary realizations of an LSTM cell's components (Fig. 3). Next, we discuss these components and their integration into a complete network (Fig. 1).

## *A. Stochastic vs. Binary LSTM Cell Components*

As can be seen in Fig. 3, an LSTM consists of three components called forget, input and output gates, several multipliers, an adder, and an activation function (*AF*). All these components can be implemented either using SC or binary primitives. In hybrid architectures, stochastic-to-binary (S2B) and binary-to-stochastic conversion is needed. In the following, we discuss the function and realization of the various components.

The **forget gate** determines what information will get removed from the cell state. The previous hidden state $h_{t-1}$ and the current input $x_t$ are multiplied with the corresponding weights and the bias is added before passing through a sigmoid function. When the result is 0, or close to it, that information is discarded; if it is 1 or close to 1 then it is kept. This operation is shown in Eq. 1:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{1}$$

The binary implementation of the forget gate consists of an 8-bit adder, an 8-bit multiplier, and a look-up table implementation of the sigmoid function. The SC version uses an XNOR gate for multiplication and a MUX for scaled addition as in Fig. 2. A small finite-state machine (FSM) [2] is used for the sigmoid function; note that this function processes scaled outcomes of the addition.

Next, the **input gate** determines what new information is added to the cell state. The previous hidden state $h_{t-1}$ and the current input $x_t$ are passed through a sigmoid function. This transforms the values between 0 ("not important") and 1 ("important") and determines how much impact the new information has on the cell output. The values $h_{t-1}$ and $x_t$ are also passed through an *AF*, which helps to regulate the network. *AF* is usually the tanh function, but in the next section, we will also evaluate the ReLU function for our stochastic implementation. Note that *AF* (tanh or ReLU) is not the same function as the sigmoid σ. Two intermediate values $i_t$ and $C'_t$ are produced (Eqs. 2 and 3):

$$i_t = \sigma\,( W_i \cdot [h_{t-1}, x_t] + b_i ) \tag{2}$$

$$C'_t = AF\,( W_c \cdot [h_{t-1}, x_t] + b_c ) \tag{3}$$

Finally, the **output gate** determines the output value $o_t$. The current input and the previous hidden state are passed through the sigmoid function (Eq. 4):

$$ot = \sigma\,(W_o \cdot [h_{t-1}, x_t] + b_o ) \tag{4}$$

The new cell state $c_t$ and hidden state $h_t$ are then calculated using the values produced by the three gates. $c_t$ is obtained by multiplying the previous cell state $c_{t-1}$ with the forget gate output $f_t$. If the forget gate value is 0, then the previous cell state value is dropped, otherwise, this value is added to the

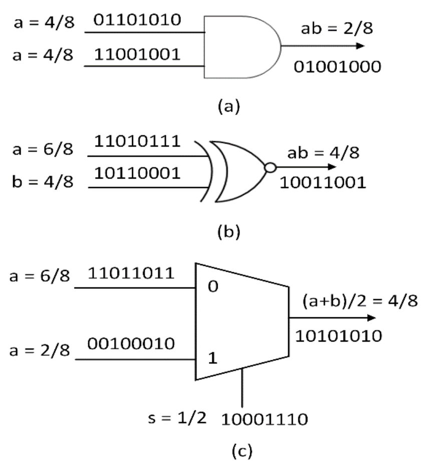

**Figure 2:** Unipolar multiplication (a), bipolar multiplication (b), and scaled addition using a multiplexer (MUX) (c).

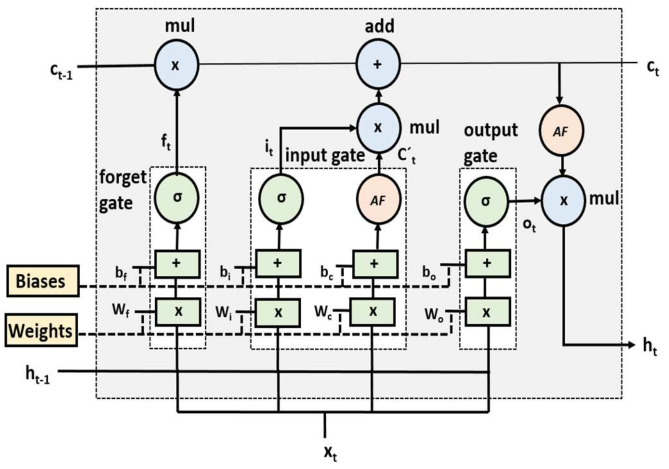

**Figure 3:** LSTM cell with input $x_t$, cell state $c_{t-1}/c_t$, hidden state $h_{t-1}/h_t$, internal signals $f_t$, $i_t$, $C'_t$, $o_t$ weights $W_f$, $W_i$, $W_c$, $W_o$, biases $b_f$, $b_i$, $b_c$, $b_o$, activation function *AF.*

output of the input gate and the cell state value is updated to $c_t$ (Eq. 5). This value is passed through *AF* and then multiplied with $o_t$ to get the new hidden state (Eq. 6). $h_t$ and $c_t$ are passed to the next time step.

$$c_t = f_t \cdot c_{t-1} + i_t * C'_t \tag{5}$$

$$h_t = \sigma \cdot AF(c_t) \tag{6}$$

Like the forget gate, the input and output gates and the arithmetic operations in Eqs. 5 and 6 can be implemented using binary or stochastic primitives. One goal of our investigations in this paper is to find the best realization of the activation function *AF*, which we discuss next.

## *B. Stochastic Realizations of the Activation Function*

Activation functions (*AF* in Fig. 3 and Eqs. 3 and 6) are key operations in LSTMs. They have a significant impact on the performance of the NNs [15] [16] and therefore, they have to be carefully selected. We now consider the two main alternative activation functions tanh and ReLU.

The most widely used activation function for LSTMs is tanh. The *stochastic tanh* function is often implemented by a special saturating-counter FSM with *N* states [2]. It outputs 0 (1) when the current state is in the first (last) *N*/2 states (Fig 4b). But the tanh function has drawbacks. It saturates to 1 (−1) for larger (smaller) values and is only strongly sensitive to its inputs when it is near 0. Thus, the gradient of tanh can be very small. In an LSTM, such gradients can get multiplied over many time steps, resulting in an overall gradient that diminishes exponentially. This leads to the vanishing-gradient problem discussed above. Therefore, we also investigate ReLU as an alternative activation function.

The *rectified linear unit* or *ReLU* is defined in Eq. 7:

$$RELU(x) = max\ \{0, x\} \tag{7}$$

In our architectures, we used an SC implementation of ReLU that is based on the exact stochastic max function from [17] (Fig. 4a). It helps to solve the issue of vanishing gradient faced by tanh, because its gradient tends to be well controlled and remains proportional to the node activations (1 for positive values and 0 for negative values). So even when the gradients are multiplied over time, they do not vanish and therefore do not reduce the learning capability of the network. ReLU computations are also easier as they do not involve exponential functions. Furthermore, ReLU can lead to desirable sparse representations, because ReLU is 0 for all negative values and is rarely active [18]. This means that not all neurons have to be active for the network to operate. This results in better predictions and less overfitting.

## *C. LSTM Cell Architectures*

To fully understand the potential of SC for lightweight hardware realizations of LSTM networks, we compare four architectures of individual LSTM cells. Table I provides an overview of the architectures, whereas Fig. 5 depicts their schematics. For the reader's convenience, we include the symbols ■, ▼,▲, ◆ used in Fig. 6 to distinguish the four architectures.

- ***FB* (fully-binary,■)** has all components of an LSTM cell implemented in binary. It does not need any SNG or S2B blocks and serves as a baseline for the other architectures.
- ***HBS* (hybrid binary-stochastic,▼)** has the *AF* module and the three gates implemented in SC, whereas adders and multipliers "add" and "mul" are in binary. As can be seen in Fig. 5b, SNG and S2B

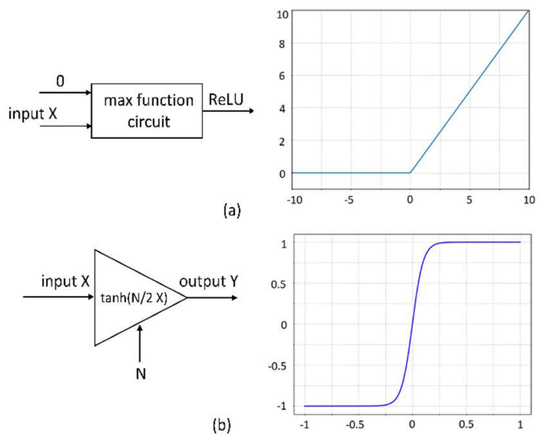

**Figure 4**: Activation functions: ReLU following [17](a), and stochastic tanh following [2] (b).

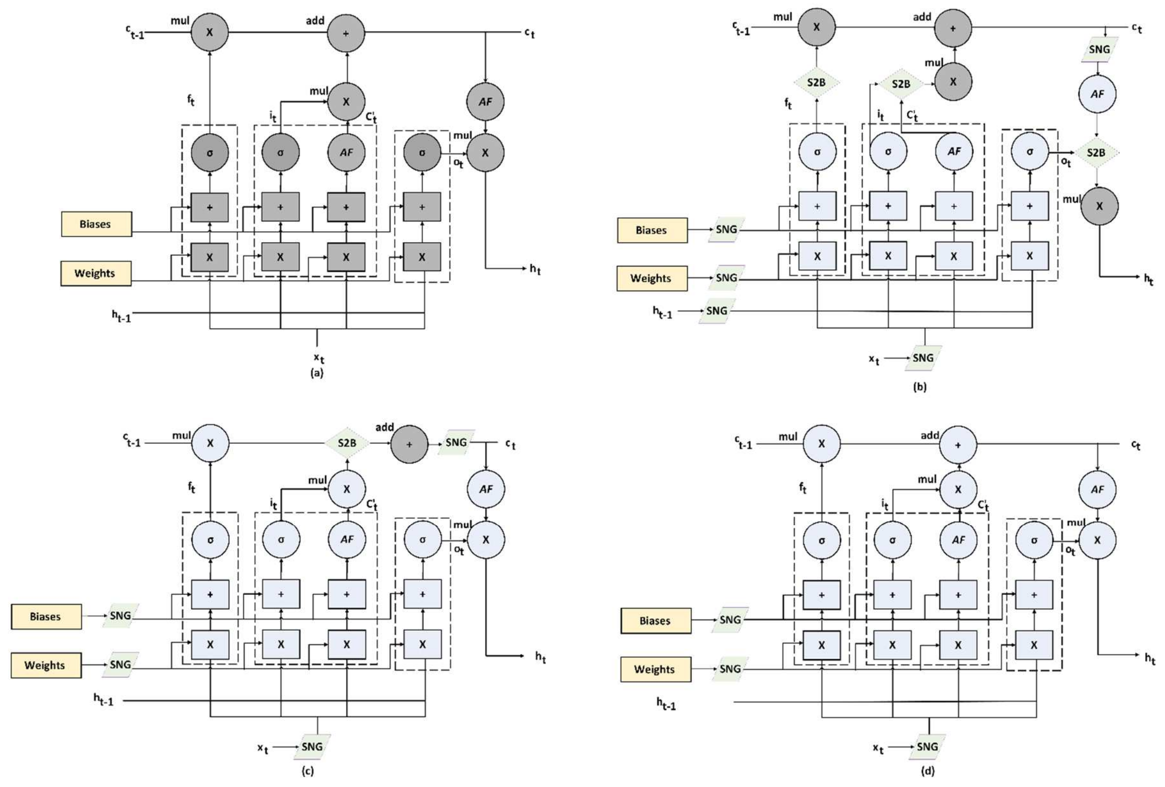

**Figure 5**: LSTM cell architectures *FB* (a), *HBS* (b), *HSB* (c) and *FS* (d). Binary components are shown in dark grey, stochastic modules are shown in light blue (see Table I). SNGs and S2Bs are colored in light green.

blocks are needed at the interface between the stochastic and binary parts.

- ***HSB* (hybrid stochastic-binary, ▲)** has all modules that were stochastic in *HBS* and, in addition, “mul” implemented in SC, while binary full adders are used for “add”.
- ***FS* (fully-stochastic,◆)** as all the modules designed in SC, including “add” based on a multiplexer. It requires SNGs at the cell’s inputs but not within the cell.

We denote a complete LSTM network based on an LSTM cell of architecture $A \in \{FB, HBS, HSB, FS\}$ by $A^{N_i,N_o}_{N_{l1},\,N_{l2}}$, where $N_i$ and $N_o$ stand for the number of neurons in the network’s input and output layers and $N_{l1}$ and $N_{l2}$ for the number of LSTM cells in one or, optionally, two hidden layers.

For example, $FB^{28,10}_{100,200}$ stands for a network with 28 input neurons, two hidden layers with 100 and 200 LSTM cells, and 10 output neurons, where each LSTM cell has architecture *FB*. $HSB^{10,2}_{128}$ describes a smaller network with 10 input neurons, one hidden layer of 128 LSTM cells of architecture *HSB*, and 2 output neurons. Note that $N_i$ and $N_o$ are determined by the problem being solved: $N_i$ is the dimensionality of the inputs being classified and $N_o$ is the number of classes. In contrast,

TABLE I. OVERVIEW OF STOCHASTIC VS. BINARY CELL COMPONENT IMPLEMENTATION IN LSTM CELL ARCHITECTURE

| LSTM cell architecture | Symbol (Fig. 6) | Forget, input, output gate | *AF* | Mul | Add |
|---|---|---|---|---|---|
| *FB* | ■ | binary | binary | binary | binary |
| *HBS* | ▼ | SC | SC | binary | binary |
| *HSB* | ▲ | SC | SC | SC | binary |
| *FS* | ◆ | SC | SC | SC | SC |

$N_{l1}$ and $N_{l2}$ can be chosen by the designer: it is possible to use a larger or a smaller network for the same classification problem. Here, a network with more hidden layers and more LSTM cells per layer will usually provide better classification accuracy, but it will be more expensive to train and use.

The SC designs considered here are meant for resource constrained edge and near-sensor systems, and therefore their focus is on relatively small, i.e., lightweight, networks not incorporating advanced techniques such as attention-based learning [19]. For this reason, we restrict ourselves to networks with up to two hidden layers.

## IV. Experimental Results

We created LSTM network circuits for five different datasets that are representative of relatively simple tasks expected in edge and near-sensor computing: MNIST, TIMIT, Japanese vowel (JV), IMDB and time series classification (TSC). For each of them, we designed four versions for the four architectures *FB, HBS, HSB, FS*. We picked parameters $N_i$ and $N_o$ based on the classification problem and $N_{l1}$ and $N_{l2}$ based on the typical values used in literature. For example, the architectures used to classify the MNIST dataset were of shape $A^{28,10}_{100,200}$, i.e., 28 input neurons, two hidden layers with 100 and 200 LSTM cells, 10 output neurons. We stress that our goal is not to provide better networks for known classification problems but to benchmark SC and hybrid realizations of existing networks against the traditional fully binary version.

We trained all architectures using the Keras software assuming binary representation and evaluated them by applying the circuits to perform inference. For architecture *FB*, we used the trained weights and biases directly, for stochastic and hybrid architectures, weights and biases were converted to SNs. For our hardware realizations, we estimated area by summing the area of the basic components (LSTM cells, neurons, SNGs etc.) computed by Synopsys Design

TABLE II. COMPARISON BETWEEN DIFFERENT SN LENGTH

| SN length | Area [mm$^2$] | Latency [µs] | Accuracy [%] |
|---|---|---|---|
| 256 bits | 0.30 | 0.47 | 87 |
| 512 bits | 0.32 | 0.94 | 89.9 |
| 1024 bits | 0.36 | 1.88 | 90.6 |

Compiler with TSMC's 28-nm library. We performed an architecture-level analysis of signals that are mutually uncorrelated and shared SNGs among such signals. The power consumption was estimated using the Synopsys tool.

To determine a suitable SN length, we generated results for SN lengths 256, 512 and 1024. A comparison between them for dataset TSC using the fully stochastic architecture $FS_{200}^{100,2}$ is presented in Table II. We found similar results for *FS* with other datasets/networks. The latency increases by a factor of 2 as we move from 256 to 512 to 1024 bits. There is a 3% accuracy improvement when 512-bit SNs are used in place of 256-bit SNs, but only 0.5% when going to 1024 bits. Therefore, we chose an SN length of 512 for our investigations. Note that relatively long SNs are acceptable in many applications such as those that are not time-critical. In other applications, the stochastic circuitry is so simple that the resulting system can satisfy the real-time constraints even for long SNs. (For instance, real-time operation was reported in [4] for the same SN length of 512).

The results on area reduction, power consumption and accuracy are summarized in Table III. It can be seen that significant area and power savings of up to 47% and 86% respectively are possible by switching to all-SC. 30-40% reduction in area and 50-70% reduction in power are achieved by hybrid architectures in most cases. We observe a drop in accuracy of 7.4% on average for the fully stochastic architecture and 3% and 5% on average for both hybrid versions. These outcomes span a design space where area and power savings can be traded for classification accuracy. Fig. 6 visualizes this design space: a designer can pick a point based on his or her priorities. The SN length (Table II) defines a third dimension of the design space. As expected, longer SNs mean longer run times and slightly higher area, but also higher accuracy.

There are only two prior stochastic RNN implementations that are directly comparable with our work. SCRNN [6] reports results for two of our five datasets: Japanese Vowels (achieving 29% area reduction, 55% power reduction and 93.8% accuracy) and TIMIT (15% area reduction, 83% power reduction and 71.9% accuracy). These numbers are outperformed by our hybrid designs with respect to one of the two parameters; however, the authors of [6] used shorter SNs. The RNN implementation based on a different "sign-magnitude" SN encoding [7] was reported to achieve 99% accuracy on the MNIST dataset, but the SNG requirements of this case led to an area overhead of more than 100% (instead of saving area, 11.2% more area is required). Fig. 6 includes approximate data points for "SCRNN" [6] and "SM-SCRNN" [7] (marked as "Earlier designs" and the symbol ●). We do not include a detailed comparison with [20], which was designed for FPGAs and used FPGA-specific features. Our rough estimates indicate that our direct hardware implementation requires approximately a third less area than that design.

Next, we studied the impact on accuracy achieved when using two different activation functions: tanh and ReLU. Fig. 7 compares the accuracy differences (%) of the four LSTM cell-architectures for each of the five datasets when we replace

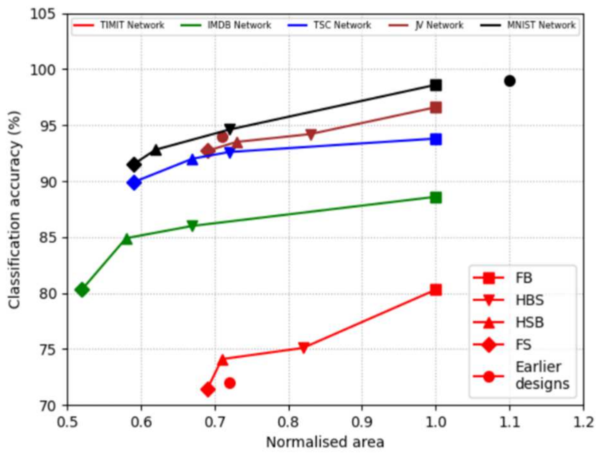

**Figure 6**: LSTM design space exploration and comparison with SCRNN [6] and SM-SCRNN [7] (both are represented as "Earlier designs"). This shows FS has clear advantage over other LSTM architectures in terms of area.

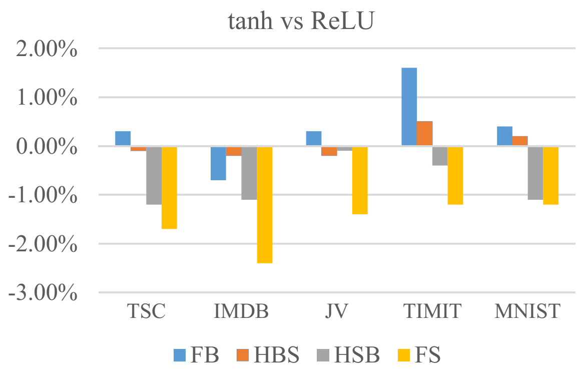

**Figure 7**: Differences in classification accuracy (%) of the four LSTM cell-architectures for each of the five datasets, when tanh is replaced by ReLU AF.

tanh with ReLU as *AF*. The accuracy difference (tanh – ReLU %) shows that consistent with conventional wisdom, tanh performs better than ReLU in full-binary architecture *FB*, with one exception (IMDB). However, this trend reverses for stochastic and hybrid architectures *HSB* and *FS*, where ReLU is consistently better than tanh. Both functions lead to very similar results for hybrid architecture *HBS*. For example, in case of TIMIT, using tanh instead of ReLU as *AF* results in an increase of accuracy (~1.5% for FB and ~0.5% for HBS). However, for HSB and FS, the accuracy of the architectures with tanh *AF* is less than the architectures with ReLU *AF* by ~0.4% and 1.2% respectively. This is consistent with the finding of [21][22] that FSM-based tanh can have a high correlation between the outputs in the consecutive clock cycles, leading to longer latencies and higher variance for inputs close to 0. Both mechanisms lead to small errors that tend to accumulate and amplify in LSTMs with their recurrent structure. This problem does not occur in ReLU, where the circuit of [17] produces exact maximum values and does not introduce any variance even for large LSTM networks.

We also re-estimated the area reduction when ReLU is used instead of tanh and found that it improves by a small amount (1 to 2%). Overall, ReLU based on a good SC implementation of the maximum function appears to be a marginally better activation function for SC LSTMs.

## V. CONCLUSION

We have demonstrated here for the first time that fully stochastic LSTMs can deliver high accuracy despite

TABLE III. Experimental Results for ***FB***, ***HBS***, ***HSB*** and ***FS***.

| Dataset | LSTM network architecture $A^{N_i,N_o}_{N_{l1},N_{l2}}$ | Area reduction compared to *FB* [%] | | | Power consumption reduction [%] | | | Classification accuracy [%] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *HBS* | *HSB* | *FS* | *HBS* | *HSB* | *FS* | *FB* | *HBS* | *HSB* | *FS* |
| TSC | $A^{100,2}_{200}$ | 28 | 34 | **43** | 57 | 77 | **85** | 93.8 | 92.6 | 92 | **89.9** |
| IMDB | $A^{10,2}_{128}$ | 35 | 43 | **47** | 61 | 79 | **86** | 88.6 | 86.0 | 84.9 | **80.3** |
| JV | $A^{12,9}_{150}$ | 17 | 27 | **32** | 50 | 71 | **76** | 96.6 | 94.2 | 93.5 | **92.7** |
| TIMIT | $A^{12,10}_{300,300}$ | 15 | 29 | **31** | 48 | 75 | **84** | 80.3 | 75.1 | 74.1 | **71.4** |
| MNIST | $A^{28,10}_{100,200}$ | 27 | 38 | **41** | 52 | 73 | **79** | 98.6 | 94.8 | 93.5 | **92.7** |

accumulated errors. A designer looking for a lightweight LSTM can now choose among more area efficient (up to 47%), more power efficient (up to 86%), and more accurate realizations. These findings extend the range of SC LSTM by including a novel fully stochastic and various hybrid versions. We also observed that ReLU is a more suitable activation function than the usual tanh in stochastic LSTMs.

## VI. References

[1] A. Alaghi, W. Qian, and J.P. Hayes, "The promise and challenge of stochastic computing", *IEEE Trans. on CAD of Integrated Circuits and Systems* 37(8): 1515-1531, 2018.

[2] B. D. Brown and H. C. Card, "Stochastic neural computation I: Computational elements", *IEEE Trans. Computers* 50: 891-905, 2001.

[3] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han, "A survey of stochastic computing neural networks for machine learning applications", *IEEE Trans. Neural Networks and Learning Systems*. 32(7): 2809-2824, 2021.

[4] P. Kelettira Muthappa, F. Neugebauer, I. Polian, and J. P. Hayes, "Hardware-based fast real-time image classification with stochastic vomputing", *Proc. ICCD*, 340-347, 2020.

[5] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, 1997.

[6] Y. Liu, L.Liu, F. Lombardi, and J. Han, "An energy-efficient and noise-tolerant recurrent neural network using stochastic computing", *IEEE Trans.VLSI Systems*, *27*(9):2213-2221, 2019.

[7] A. Zhakatayev, S. Lee, H. U. Sim, and J. Lee, "Sign-magnitude SC: getting 10X accuracy for free in stochastic computing for deep neural networks", *Proc. DAC*, 1-6, 2018.

[8] F. Neugebauer, I. Pollian, and J.P. Hayes, "Framework for quantifying and managing accuracy in stochastic circuit design", *Proc. DATE*, 1-6, 2017.

[9] W.J. Gross, V.C. Gaudet and A. Milner, "Stochastic implementation of LDPC decoders", *Proc. 39th Asilomar Conf. Signals, Syst. Comput.,* Oct./Nov. 2005.

[10] P. Li and D. J. Lilja, "Using stochastic computing to implement digital image processing algorithms", *Proc.ICCD*, Oct. 2011.

[11] Y. N. Chang and K. K. Parhi, "Architectures for digital filters using stochastic computing", *Proc. ICASSP*, 2013.

[12] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks", *Proc. ICASSP*, 2013.

[13] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks", *Proc. NIPS*, vol. 2, Dec. 2014.

[14] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult", *IEEE Trans. on Neural Networks*, 1994.

[15] W. Duch, N. Jankowski, "Survey of neural transfer functions", *Neural Comput Surv* 2:163–213, 1999.

[16] G.S.S. Gomes, T.B. Ludermir, L.M.M.R. Lima, "Comparison of new activation functions in neural network for forecasting financial time series", *Neural Comput Appl* 20:417–439, 2010.

[17] F. Neugebauer, I. Polian, J.P. Hayes, "On the maximum function in stochastic computing", *Proc. Comp. Frontiers*, 59-66, 2019.

[18] X. Glorot, A. Bordes, Y. Bengio, "Deep sparse rectifier neural networks", *Proc. 14th Conf on Artificial Intelligence and Statistics*, in PMLR 15:315-323, 2011.

[19] A. Galassi, M. Lippi, P. Torroni, "Attention, please! A critical review of neural attention models in natural language processing", *CoRR abs/1902.02181,* 2019.

[20] G. Maor, X. Zeng, Z. Wang, Y. Hu, "An FPGA implementation of stochastic computing-based LSTM", *Proc. ICCD* 2019.

[21] C. Ma and D. J. Lilja, "Parallel implementation of finite state machines for reducing the latency of stochastic computing", *Proc. ISQED*, 2018.

[22] R. Goot, I. Levin and S. Ostanin, "Fault latencies of concurrent checking FSMs", *Euromicro Symp. on Digital System Design. Architectures, Methods and Tools*, 2002.