Using Stochastic Computing to Reduce the Hardware Requirements for a Restricted Boltzmann Machine Classifier

Bingzhe Li, M. Hassan Najafi, and David J. Lilja
Department of Electrical and Computer Engineering
University of Minnesota, Twin Cities, USA
{lixx1743, najaf011, lilja}@umn.edu

ABSTRACT

Artificial neural networks are powerful computational systems with interconnected neurons. Generally, these networks have a very large number of computation nodes which forces the designer to use software-based implementations. However, the software based implementations are offline and not suitable for portable or real-time applications. Experiments show that compared with the software based implementations, FPGA-based systems can greatly speed up the computation time, making them suitable for real-time situations and portable applications. However, the FPGA implementation of neural networks with a large number of nodes is still a challenging task.

In this paper, we exploit stochastic bit streams in the Restricted Boltzmann Machine (RBM) to implement the classification of the RBM handwritten digit recognition application completely on an FPGA. We use finite state machine-based (FSM) stochastic circuits to implement the required sigmoid function and use the novel stochastic computing approach to perform all large matrix multiplications. Experimental results show that the proposed stochastic architecture has much more potential for tolerating faults while requiring much less hardware compared to the currently un-implementable deterministic binary approach when the RBM consists of a large number of neurons. Exploiting the features of stochastic circuits, our implementation achieves much better performance than a software-based approach.

Keywords

Neural network; Restricted Boltzman Machine; FPGA-based implementation; stochastic computing.

1. INTRODUCTION

With growing interest in neural networks, new methods are needed to achieve low-power back-end applications or high performance neural networks rather than being restricted to software implementations running on general-purpose computing systems. However, the capabilities of neural networks highly depend on the size of the network. Power consumption and resource utilization are now considered the main limitations in implementing large neural networks in hardware. The Restricted Boltzmann Machine (RBM) is a type of artificial neural network that is capable of representing and solving difficult problems. Like other machine learning models, the RBM has two process types: learning and testing. During learning, the system is presented with a large number of input examples and desired outputs to generate a suitable RBM structure which learns a general rule to map inputs to outputs. During testing, the RBM produces outputs for new inputs following the general rule obtained in the learning process [6]. The structure presented in Figure 1 is an example of a typical two-layer RBM structure.

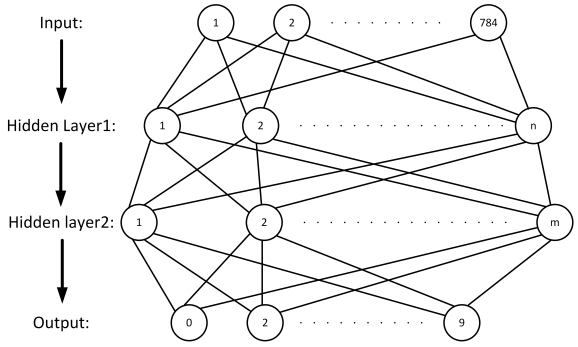

Figure 1: Two-layer RBM structure

Generally, neural networks such as the RBM require enormous and complex computations since parts of their computations, such as the sigmoid function ($Y = 1/(1 + e^{-x})$), do not have a straightforward hardware implementation. Previous work often uses the traditional binary radix encoding to implement parts of the RBM in hardware while other parts are computed on a host processor [11]. However, none of this prior work could implement a large neural network in a single FPGA due to the complexity of the hardware implementation of their methods. In this paper, we exploit the unique characteristics of stochastic computing to implement a large RBM for classifying the most well-known handwriting digit image recognition data set, MNIST [5], completely in one FPGA. The previously proposed structures require substantially more hardware resources than what today's best FPGAs provide. The proposed architecture can reduce the hardware resource requirements significantly to the point that it can be implemented completely on current FPGAs. We extend the work previously presented in [6] by

FPGA '16, February 21–23, 2016, Monterey, CA, USA.

DOI: http://dx.doi.org/10.1145/2847263.2847340

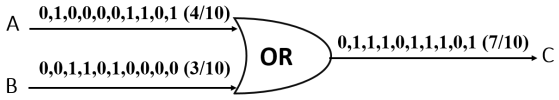

Figure 2: Example of stochastic addition using an OR gate

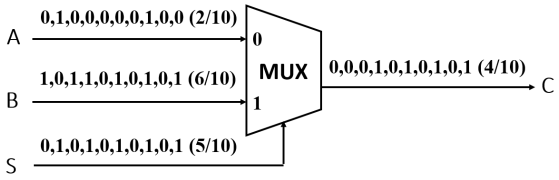

Figure 3: Example of stochastic addition using MUX

providing experimental results on the fault tolerant capability of the proposed stochastic RBM. We quantify hardware resource requirement, timing, and classification error rate of the stochastic RBM when shrinking the network size. Comparing with the software-based approach we show that the stochastic implementation achieves much better performance than the software-based MATLAB implementation.

2. STOCHASTIC COMPUTATIONAL ELEMENTS

2.1 Stochastic Bit Streams

A stochastic bit stream is a sequence of binary digits in which the information is contained in the primary statistic of the bit stream or the probability of any given bit in the stream being a logic '1'. The two encoding formats of stochastic bit streams, unipolar and bipolar [1], can express a real number x in the intervals [0, 1] or [-1, 1], respectively. In the unipolar format, the probability of having '1's in a bit stream X is $Pr(X) = x$. In the bipolar format, the probability of seeing '1's in a bit stream X is $Pr(X) = (x+1)/2$ [7]. For the values out of [-1, 1] and [0, 1], the equations shown in (1) could be used to scale the inputs to the required interval using a scaling coefficient of N.

$$Pr(X) = x/N \qquad Pr(X) = (x/N + 1)/2 \tag{1}$$

2.2 Stochastic Operations

Dickson et al [2], and Qian et al [9] introduced two methods of adding stochastic bit streams. Dickson et al used a standard OR gate to approximate addition as illustrated in Figure 2. As can be seen, the output of the OR gate, 7/10, is the expected output from adding the input values, 4/10 and 3/10. As shown in equation (2), performing an add operation using an OR gate introduces an extra AB term which is an error in the result. The only case that the AB term can be ignored is when A $\ll 1$ and B $\ll 1$.

$$C = A \; or \; B = A + B - AB \tag{2}$$

Qian et al implemented scaled addition using a MUX. According to equation (3), when the select line of a MUX is a representation of 0.5 in the stochastic domain, the output is (A+B)/2 which scales down the desired result by a factor of two. Figure 3 shows an example of performing scaled addition using the MUX.

$$C = A \cdot S + B \cdot (1 - S) \tag{3}$$

To generalize the simple two-input adder to a multi-input adder, equations (2) and (3) can be extended to (4) and (5).

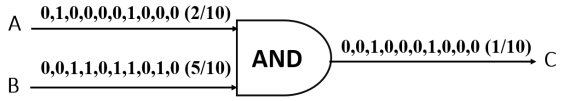

Figure 4: Example of Stochastic multiplications using an AND gate

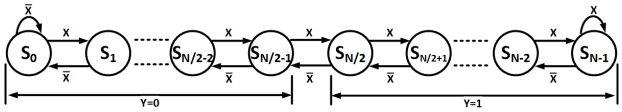

Figure 5: State transition diagram of the FSM-based stochastic tanh function

$$C = A_1 \; or \; A_2 \; or \; A_3 \; or \; ... \; or \; A_N \tag{4}$$

$$C = \frac{1}{N} \sum_{i=1}^{N} A_i \tag{5}$$

In the stochastic domain, multiplication could be implemented using a standard AND gate for the unipolar (Figure 4) and an XNOR gate for the bipolar representation of bit streams [7]. Brown and Card [1] proposed a stochastic implemention of the hyperbolic tangent (tanh) function using a finite state machine (FSM) (Figure 5). The FSM presented in Figure 5 implements equation (6) where N is the number of states in the FSM, and X and Y are the input and output stochastic streams in bipolar format.

$$Y = tanh(N/2)X \tag{6}$$

Note that N/2 (N>2) in equation (6) is also the coefficient of the input which increases the value of the input by a factor of N/2 during computation of the tanh function. Since we are looking for the result of tanh(X) rather than tanh(N/2*X), the N/2 coefficient is a potentially unwanted scaling factor. In the next section, we explain how to take advantage of this N/2 coefficient to obtain the desired results.

3. STOCHASTIC IMPLEMENTATION

Handwritten digit recognition is an application of the RBM neural network. In this paper, the RBM coefficents required for handwritten digit recognition are first generated based on the Hinton et al [3, 10] learning code and then the testing step will be achieved completely on an FPGA-based hardware platform. The selected deep BM structure (Figure 1) is a 2-layer BM in which the inputs to the first layer are thousands of 28×28 pixel images provided by the MNIST data set. The first and the second hidden layers have n and m neural units, respectively. The maximum value produced at the ten units of the output layer determines the final handwritten decimal digit result. Equations (7), (8) and (9) show the functions of the different RBM layers [10].

$$\mathbf{w1p} = 1/(1 + \exp(\mathbf{data} * \mathbf{w1\_vishid} + \mathbf{coef})) \tag{7}$$

$$\mathbf{w2p} = 1/(1 + \exp(-\mathbf{w1p} * \mathbf{w2} - \mathbf{bias\_pen})) \tag{8}$$

$$\mathbf{out} = \exp(\mathbf{w2p} * \mathbf{w\_class} + \mathbf{bias\_top}) \tag{9}$$

where the **data** (1×784) matrix is the handwritten input image and $\mathbf{coef} = \mathbf{temp\_h2} * \mathbf{w1\_penhid} + \mathbf{bias\_hid}$. Note that **w1_vishid** ($784 \times n$), **temp_h2** ($1 \times m$), **w1_penhid** ($m \times n$), **w2** ($n \times m$), **bias_hid** ($1 \times n$), **bias_pen** ($1 \times m$), **w_class** ($m \times 10$) and **bias_top** (1×10) are the RBM

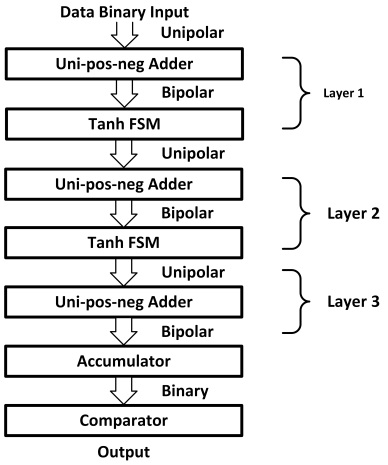

Figure 6: The RBM bit stream implementation data flow

constants computed by the learning process. **w1p** and **w2p** are the probabilities of the first and second hidden layers.

To reduce the complexity of the computations and thereby save FPGA resources, we simplify these equations in two steps. First, we precompute constant coefficients generated by the learning phase For example, $\mathbf{coef} = \mathbf{temp\_h2} * \mathbf{w1\_penhid} + \mathbf{bias\_hid}$ is precomputed so that it is necessary to store only the constant value **coef** into the FPGA memory. Secondly, our main goal is to find the maximum value of the 10 outputs produced at the output layer. The purpose of the exponential function in (9) is simply to make finding the maximum value easier by enlarging the differences between the outputs. Thus, the exponential function is actually redundant and we can rewrite equation (9) and convert it to equation (10).

$$\mathbf{out} = \mathbf{w2p} * \mathbf{w\_class} + \mathbf{bias\_top} \tag{10}$$

3.1 RBM Data Flow

As seen in equations (7), (8) and (10), the RBM layers use some similar operations including multiplications and additions. Furthermore, equations (7) and (8) use the same function format, $1/(1+exp(x))$. However, these similarities do not mean that we can directly connect different layers of the RBM to each other in the stochastic architecture since different stochastic computation elements might use different encoding formats in their inputs and outputs. Figure 6 shows the implementation dataflow of the proposed stochastic RBM. As can be seen in this figure, the input image, which is in unipolar format, comes into a *Uni_pos_neg* adder (see Section 3.3) while its output in bipolar format goes to a bipolar stochastic FSM-based tanh module. The output of the tanh module is in unipolar format which fits with the next layer input and so forth. Although we have used different encoding formats in the structure, each function has been designed in a way that produces its output compatible with the required format for the input of the next level.

Note that we scale down all inputs by an N/2 factor to make sure that all values (inputs, constant coefficients and inter-media results) belong to [-1,1] or [0,1] in order to be able to represent them by stochastic bit streams. This N/2 scaling is canceled automatically by the multiplication with N/2 in equation (6) while it still provides us with the advantage of reducing the errors in equations (2) and (4).

3.2 Matrix Multiplication

A simple presentation of the matrix multiplication performed in the RBM is shown in (11) where each element in the result matrix C is calculated according to equation (12).

$$\mathbf{C} = \begin{bmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{bmatrix} \times \begin{bmatrix} B_{11} & \cdots & B_{1k} \\ \vdots & \ddots & \vdots \\ B_{m1} & \cdots & B_{mk} \end{bmatrix} \tag{11}$$

$$C_{ij} = \sum_{s=1}^{m} A_{is} \cdot B_{sj} \tag{12}$$

Based on equation (12), the element C_{ij} of the matrix multiplication result in (11) is the sum of the multiplications of elements A_{is} and B_{sj}. In the stochastic domain, multiplication can be implemented easily by a simple two-input AND gate. However, for the addition operations required in computing each element of the result matrix, **C**, the number of add operations is proportional to the size of the RBM layers.

Taking an example, $\mathbf{data} * \mathbf{w1\_vishid}$, in equation (7), if the input image size is 28×28, a $[1 \times 784] \times [784 \times n]$ matrix multiplication needs to be performed using stochastic circuits, which consists of 784 addition operations. As discussed in Section 2, there are two types of stochastic adders, MUXs and ORs. The N-input scaled adder defined in equation (5) always scales the result down N times and the OR gate introduces an extra error with the term AB in equation (2). Implementing such a large number of additions seems to be a challenge for both the MUX and the OR gates.

For the MUX gate, consider the example presented in Figure 7 in which the expected result of adding 784 input values is 0.8 while we want to use 1024 length bit streams to perform stochastic addition. In the ideal case, when a MUX is being used as the stochastic adder and there is not any correlation between input bit streams, the stochastic result is about 0.8/784=0.00102. This value is too small to be represented even with 1024-bit streams because the output stream will only have a single '1' out of 1024 bits as shown in Figure 7. Representing such small values can make the system vulnerable even to a very small number of errors.

For the OR gate, in order to perform the stochastic addition accurately, we scale the inputs of the OR gate down to properly small values (in our case, scale factor N/2=16) to be able to ignore the effect of the extra AB term in (2) and to avoid the large soft error like MUX adder. For example, if the result of 0.4+0.4 is going to be computed directly using an OR gate, it could introduce about 0.4*0.4/(0.4+0.4)=20% error just from the AB term. However, if we first scale the inputs down 10 times and then scale the result back, the error rate reduces from 20% to 0.04*0.04/(0.04+0.04)=2%. From the soft error point of view, assume that the result of adding 784 input values using the OR gate is 0.08 which contains $1024 * 0.08 \approx 82$ '1's in a 1024-bit stream. If an error causes one bit of the stream to flip from '0' to '1', the output error rate will be just $(83/1024 - 0.08)/0.08 \approx 1.32\%$. Therefore, an OR gate with a proper scaling factor for inputs can be a suitable addition circuit that can help the RBM structure to experience very small error rates.

The default RBM size in our implementations is $500 \times 1000 \times 10$ which means that the first hidden layer of the BM has 500 units, the second hidden layer has 1000 units, and finally the output has 10 units. Selecting this structure, there are three massive matrix multiplications, $[1 \times 784] \times$

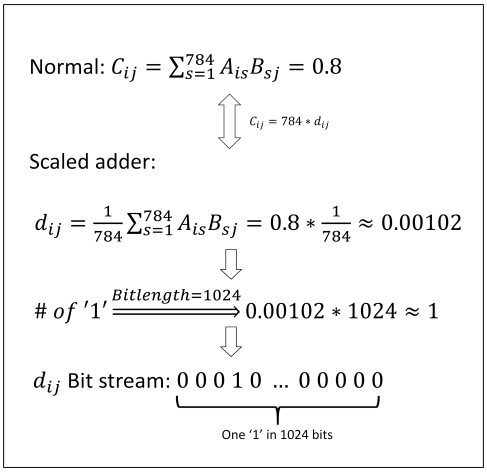

Figure 7: An example of producing outputs vulnerable to soft errors when performing stochastic scaled addition.

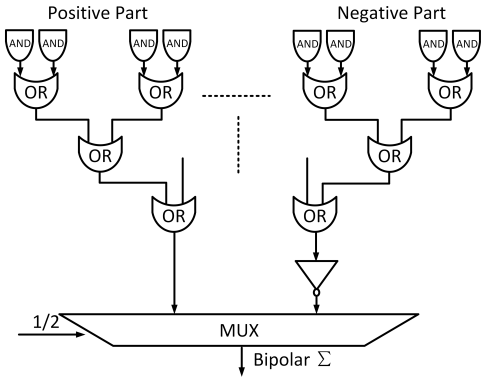

Figure 8: *Uni-pos-neg* adder: A large matrix multiplication hardware structure using both positive and negative inputs.

$[784\times500]$, $[1\times500]\times[500\times1000]$, and $[1\times1000]\times[1000\times10]$ to perform. In order to compute each element of the result matrix using stochastic circuit, we introduce the *Uni_pos_neg adder* (Figure 8).

3.3 Uni_pos_neg adder

The circuits used in the stochastic RBM use both positive and negative values. However, the unipolar representation of stochastic numbers is only suitable for representing pure positive or pure negative values. Since the stochastic tanh function accepts input bit streams in only bipolar format, the output bit streams generated from stochastic matrix multiplication needs to be also in that format. Thus, in this section we propose a *Uni_pos_neg* adder module for implementing the matrix multiplication structure which can automatically handle the format conversion process.

Considering equation (13) as an example, it computes one element of the result matrix of multiplying two matrices. The element C_{ij} in the result matrix equals the sum of the products of $data_{is}$ and $w1\_vishid_{sj}$. Since the products of $data_{is}$ and $w1\_vishid_{sj}$ contain both positive and negative values, we cannot directly use the "OR" gate as the stochastic adder. In the *Uni_pos_neg* adder, the products are first divided into two parts, negative part and positive part, based on the sign bit of the binary value, $w1\_vishid_{sj}$, located in the most significant bit (MSB). Since **data** is the input image that must be scaled to [0, 1], the sign of the product of $data_{is}$ and $w1\_vishid_{sj}$ only depends on the sign of $w1\_vishid_{sj}$. Finally, those negative and positive parts only contain negative or positive values respectively so we can simply use the stochastic arithmetic elements (AND and OR) with unipolar encoding for both parts.

For the positive part, we directly encode the binary value as a unipolar bit stream, then use AND gates to multiply $data_{is}$ and $w1\_vishid_{sj}$, and OR gates to sum up all of the positive products. For the negative part, at first we consider them as positive values (flipping the MSB to '0') and do the same operations as the positive part. Then, at the final stage of Figure 8, we invert the value computed in the negative part to unify the encoding format. In this way the results of equations (14) and (15) will be obtained after going through the addition tree structure seen in Figure 8.

$$C_{ij} = \sum_{s=1}^{m} data_{is} \cdot w1\_vishid_{sj} \tag{13}$$

$$A = \sum M_{pos} = \sum_{pos} data_{is} \cdot w1\_vishid_{sj} \tag{14}$$

$$B = \sum M_{neg} = \sum_{neg} data_{is} \cdot w1\_vishid_{sj} \tag{15}$$

In equations (14) and (15), A and B are the sums of the positive and the negative products in the matrix multiplication with unipolar encoding. We invert the negative part, B, and use a MUX adder to sum A and the inverted B as seen in Figure 8 to obtain the *sum* variable of equation (16). Comparing this variable with the bipolar expression $Pr(X) = (x+1)/2$, *sum* is the bipolar encoding of C_{ij}. The inverter and MUX automatically transfer two unipolar values, A and B, into the bipolar format version of A-B. Thus, the *Uni_pos_neg* adder produces one element of the output matrix of the matrix multiplication.

$$sum = \frac{A + (1 - B)}{2} = \frac{1 + (A - B)}{2} \tag{16}$$

For the other BM layers, according to equations (8) and (10), **w1p** and **w2p** are calculated using the sigmoid function and hence their values belong to [0, 1]. We repeat the above process for other layers' matrix multiplications by regarding **w1p** and **w2p** as **data** and dividing inputs into the positive and negative parts according to the sign of **w2** and **w_class** the same as is done for **w1_vishid**.

3.4 Sigmoid Function

To implement the sigmoid function, we make some transformations according to equations (17), (18) and (19). Based on these transformations, the sigmoid function could be considered as a bipolar encoding of the tanh function. As seen in Figure 9, both the input and the output of the tanh function are in bipolar format. However, to match with the restrictions of the RBM structure, we obtain the sigmoid function with bipolar inputs and a unipolar output based on (19).

$$tanh(x) = (e^{x} - e^{-x})/(e^{x} + e^{-x}) \tag{17}$$

$$\begin{aligned} \frac{1}{1+e^{-x}} &= \frac{e^{x/2}}{e^{x/2}+e^{-x/2}} = 1/2(1 + \frac{e^{x/2}-e^{-x/2}}{e^{x/2}+e^{-x/2}}) \\ &= (1 + tanh(x/2))/2 \end{aligned} \tag{18}$$

$$\frac{1}{1+e^{-Nx}} = \frac{1 + tanh(Nx/2)}{2} \tag{19}$$

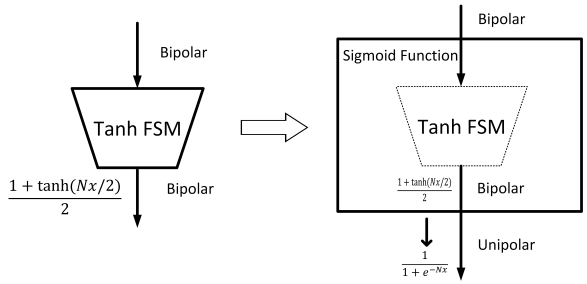

Figure 9: Sigmoid function implementation with bipolar input and unipolar output from tanh function

Table 1: Error rate comparison between the proposed stochastic architecture with different lengths of bit streams and the conventional version.

Sizes	Conventional	Stochastic			
		512	1024	2048	4096
$100 \times 200 \times 10$	1.96%	10.06%	5.72%	3.91%	3.14%
$200 \times 400 \times 10$	1.35%	7.93%	4.24%	3.08%	2.35%
$300 \times 600 \times 10$	1.17%	6.33%	2.92%	1.80%	1.80%
$400 \times 800 \times 10$	1.10%	4.17%	2.37%	1.52%	1.34%
$500 \times 1000 \times 10$	0.98%	5.10%	2.32%	1.64%	1.32%

4. EXPERIMENTAL RESULTS

The proposed stochastic architecture for the RBM has been implemented using the Verilog HDL language, then synthesized, placed and routed using the Xilinx ISE Design Suite 14.7 on the Xilinx Virtex7 xc7v2000t-2flg1925 FPGA. Functional verification of the proposed architecture has been done in Matlab where we implemented both the conventional approach [3] and also our proposed stochastic architecture to work on 10,000 MNIST handwritten test images. Since computation time and the accuracy of stochastic operations are directly proportional to the length of bit streams [9], we ran our simulations for the stochastic architecture on four different lengths of bit streams, 512, 1024, 2048, and 4096 bits, to obtain the trade-off between the output error rate and the required computation time. For converting the deterministic input values into stochastic bit streams we have used the stochastic number generator presented in [9].

4.1 Error Rate Comparison

For the conventional approach we used the MATLAB training method presented in [3] and obtained the output error rates of different RBM sizes as shown in Table 1. The reported error rates are the percentage of miss-classifications in classifying the 10,000 input test images from the MNIST data set. Based on the results presented in Table 1, the stochastic implementations, particularly the ones with the longer bit streams, could compete with the results of the conventional approach with only a little higher error rate. Based on Table 1, the error rates of the stochastic RBM decrease when increasing the length of bit streams for all different sizes of the layers. However, every additional bit in the bit stream needs one extra clock cycle to generate and process the streams. Selecting the optimum length of bit streams is up to the designer based on the application requirements and system limitations.

4.2 Fault Tolerance Comparison

An attractive feature of stochastic architectures is their ability to tolerate faults. The reason is that a single bit flip in a long bit-stream results in only a small change in the value of the stochastic number [8]. To study the fault tolerant capability of the proposed architecture we injected faults into the stochastic and also into the conventional architecture of the RBM, both implemented in the MATLAB tool. For the stochastic case, we injected the faults in the same way as [9], but in two different approaches. In the first approach we do not inject the faults into the stream generator modules whereas in the second approach the fault is injected in both internal computational elements and also the stream generators.

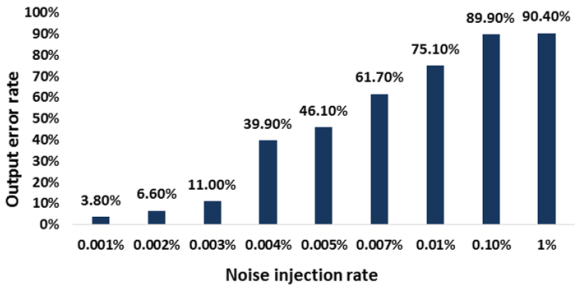

Figure 10: Output error rates obtained from injecting different rates of faults into the conventional architecture

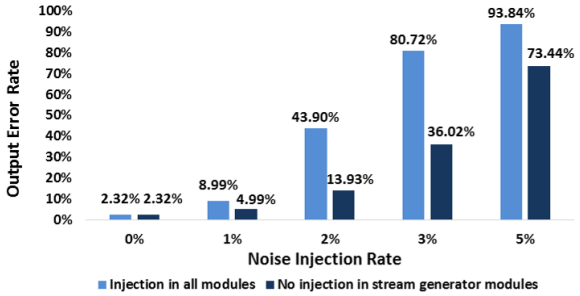

Figure 11: Output error rates obtained from injecting different rates of faults into the stochastic architecture

Figure 10 and 11 show the results of injecting faults into the conventional and stochastic implementations of the RBM. Comparing these figures, the stochastic architecture has shown a much better capability to tolerate faults. For example, in the case with 1% fault injection rate, the conventional structure produces a 90.4% error rate whereas for the stochastic implementation, in the first fault injection approach, a 4.99% error rate is reported in recognizing the handwritten digits. Considering this result, if we try to prepare more reliable stochastic stream generators, we can expect the proposed stochastic architecture to gracefully tolerate about a 1%-2% fault rate in its internal computing circuits.

4.3 Hardware Resource Comparison

As mentioned in the introduction section, implementing a fully parallel specially designed hardware implementation of a neural network in an FPGA is expensive, involves extra design overheads, and in most cases is not even possible in today's FPGAs [12]. Recently, for the first time in the literature, a fully pipelined FPGA architecture of a factored RBM has been implemented in [4]. Although the virtualized architecture proposed in [4] could implement a neural network consisting of a maximum 4096 nodes using time multiplex sharing of hardware resources in the largest overall device available at the time of that experiment (Virtex6 LX760), the largest achievable RBM without virtualization on that device is on the order of 256 nodes.

Table 2: Area, timing and error rate results for different RBM sizes.

Size	Area (# of LUTs)	Latency(s)	Output error rate(%) Stream length=1024
$100 \times 200 \times 10$	144,450	$8.561*10^{-6}$	5.72%
$200 \times 400 \times 10$	345,710	$9.682*10^{-6}$	4.24%
$300 \times 600 \times 10$	603,750	$9.797*10^{-6}$	2.92%
$400 \times 800 \times 10$	920,190	$1.025*10^{-5}$	2.37%
$500 \times 1000 \times 10$	1,292,310	$1.077*10^{-5}$	2.32%

Table 3: Comparing the latency of the software-based and the stochastic FPGA-based implementations of the RBM when the length of bit streams is 1024.

Size	CPU(s)	FPGA(s)	Speedup
$100 \times 200 \times 10$	$6.2*10^{-3}$	$8.561*10^{-6}$	724.2
$200 \times 400 \times 10$	$7.8*10^{-3}$	$9.682*10^{-6}$	805.6
$300 \times 600 \times 10$	$9.2*10^{-3}$	$9.797*10^{-6}$	939.1
$400 \times 800 \times 10$	$1.32*10^{-2}$	$1.025*10^{-5}$	1287.8
$500 \times 1000 \times 10$	$1.66*10^{-2}$	$1.077*10^{-5}$	1541.3

Implementation reports from Xilinx give us enough information about the required hardware resources for implementing the proposed stochastic architecture. As shown in Table 2, implementing the stochastic architecture of the RBM with $500 \times 1000 \times 10$ layer size needs an order of one million LUTs and could be placed and routed in Virtex7 2000 Xilinx FPGA.

4.4 Sensitivity to Network size

One solution to fit neural networks on FPGAs with fewer resources is to shrink the size of the layers while still satisfying the application expectations. In this section we explore the effect of shrinking the size of the stochastic RBM layers from the largest size (500*1000*10) that produces the lowest output error rate to smaller sizes and see how reducing the size of the layers changes the output error rate, latency and resource utilization. As shown in Table 2, by shrinking the size of the RBM, the number of LUTs is reduced dramatically while the latency does not experience much change and the error rate increases with a very slow slope.

4.5 Performance Comparison

Since the deterministic architecture of the RBM is not implementable on the current FPGAs, we analyze the performance of the proposed stochastic architecture by comparing the latency of classifying one input test image using the implemented stochastic architecture with the corresponding time for classifying by the software-based MATLAB approach. Running Hinton et al [3] code in MATLAB, we measured the latency of the software-based implementation on a machine with an Intel Core i5 Dual core CPU (3.40GHz) and 16GB DDR3 RAM. Since there are many matrix operations in the RBM application, writing appropriate programs in the MATLAB tool could be a better and more straight-forward way for running these types of applications than writing the program in C. Table 3 shows the execution time of classifying a sample image from the MNIST data set using the software-based implementation and the stochastic FPGA-based implementation for five different sizes of RBMs.

From these results, we can see that the speedup of the proposed stochastic implementation could reach to about 700x for the $100 \times 200 \times 10$ layer size and even to 1500X for the 500x1000x10 neural network size compared to the software-based MATLAB implementation. By increasing the neural network size, we obtained greater speedup because the layer size did not have much influence on the classification time of an input image in the stochastic RBM (as shown previously in Table 2), whereas larger RBM sizes cost much more computation time in the software-based implementation.

5. CONCLUSION

In this paper, we proposed a stochastic FPGA implementation of a RBM classifier using stochastic logic. In a large RBM network, matrix multiplications and sigmoid functions are the most expensive operations to be implemented. We proposed the *Uni_pos_neg* adder to implement the stochastic matrix multiplication and used the FSM tanh function to achieve the sigmoid function. Our experimental results show that the proposed stochastic architecture can tolerate 1-2% fault rate in its computing circuits. Performance evaluation results show that the proposed architecture could classify a standard handwritten input image about 700x times faster than the software-based MATLAB implementation when it is implemented on a Virtex 7 xc7v2000 FPGA.

6. ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation grant no. CCF-1408123. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

7. REFERENCES

[1] B. D. Brown and H. C. Card. Stochastic neural computation. i. computational elements. *Computers, IEEE Transactions on*, 50(9):891–905, 2001.

[2] J. A. Dickson, R. D. McLeod, and H. Card. Stochastic arithmetic implementations of neural networks with in situ learning. In *Neural Networks, 1993., IEEE International Conference on*, pages 711–716. IEEE, 1993.

[3] G. E. Hinton and R. Salakhutdinov. A better way to pretrain deep boltzmann machines. In *Advances in Neural Information Processing Systems*, pages 2447–2455, 2012.

[4] L.-W. Kim, S. Asaad, and R. Linsker. A fully pipelined FPGA architecture of a factored restricted boltzmann machine artificial neural network. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(1):5, 2014.

[5] Y. LeCun and C. Cortes. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

[6] B. Li, M. H. Najafi, and D. J. Lilja. An FPGA implementation of a restricted boltzmann machine classifier using stochastic bit streams. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 68–69. IEEE, 2015.

[7] P. Li, D. J. Lilja, W. Qian, M. D. Riedel, and K. Bazargan. Logical computation on stochastic bit streams with linear finite state machines. *IEEE Transactions on Computers*, page 1, 2012.

[8] M. H. Najafi and M. E. Salehi. A Fast Fault-Tolerant Architecture for Sauvola Local Image Thresholding Algorithm using Stochastic Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):808–812, Feb 2016.

[9] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja. An architecture for fault-tolerant computation with stochastic logic. *Computers, IEEE Transactions on*, 60(1):93–105, 2011.

[10] R. Salakhutdinov and G. Hinton. An efficient learning procedure for deep boltzmann machines. *Neural computation*, 24(8):1967–2006, 2012.

[11] M. Skubiszewski. An exact hardware implementation of the boltzmann machine. In *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*, pages 107–110. IEEE, 1992.

[12] J. Zhu and P. Sutton. FPGA implementations of neural networks–a survey of a decade of progress. In *Field Programmable Logic and Application*, pages 1062–1066. Springer, 2003.