Journal of Computer Science

Original Research Paper

Efficient Implementation of Stochastic Computing Based Deep Neural Network on Low Cost Hardware with Saturation Arithmetic

[1]Sunny Bodiwala and [2]Nirali Nanavati

[1]Department of Computer Engineering, Gujarat Technological University, Ahmedabad, India
[2]Department of Computer Engineering, Sarvajanik College of Engineering and Technology, Surat, India

Article history
Received: 27-08-2020
Revised: 21-09-2020
Accepted: 17-11-2020

Corresponding Author:
Sunny Bodiwala
Department of Computer Engineering, Gujarat Technological University, Ahmedabad, India
Email: sunny.bodiwala@gmail.com

Abstract: This study presents an efficient and rapid implementation of Stochastic Computing (SC) based Deep Neural Network (DNN) on a low-cost hardware platform. The proposed technique uses bipolar signal encoding in stochastic computing which relatively gives low hardware footprint compared to binary computing. Thereinafter, stochastic max function is presented and subsequently used to approximate the hyperbolic tangent activation function in SC. In addition, saturation arithmetic is proposed to reduce down scaling parameters that can further affect precision in computation. In this study, we demonstrate our SC-based DNN feasibility through a hardware accelerator prototype with the AXI Stream interface on a PYNQ Z2 board which is equipped with a XILINX ZYNQ XC7Z020-1CLG400C. The validity of this study is demonstrated through a MNIST handwritten digit recognition task. The experimental result shows our SC-based DNN model can be easily deployed on the embedded devices. The SC-based accelerator with AXI Stream interface performs at 1.877 GOP/s processing throughput, achieves higher accuracy with minimum area and energy consumption, consuming only 0.61 mm^2 area and 1.89W power.

Keywords: Deep Neural Network, FPGA, Accelerator, Optimization, Stochastic Computing, Custom Computing

Introduction

Humans have always dreamt of creating intelligent machines that can think. Today, Artificial Intelligence (AI) is a thriving field with many active research topics and practical applications. Humans seek to automate tasks by developing intelligent software to do daily labor work, recognize image and speech, medical diagnoses, develop virtual assistant and many more. AI systems have the capability to acquire knowledge by extracting meaningful information from raw data which is also known as machine learning (Goodfellow *et al*., 2016). Deep learning has emerged as a new area of machine learning research that allows a computer to automatically learn complex functions directly from the data by extracting representations at multiple levels of abstraction (Deng and Yu, 2014; LeCun *et al*., 2015). Deep Neural Networks (DNNs) have achieved unprecedented success in many machine learning applications such as speech recognition (Abdel-Hamid *et al*., 2014) and visual object recognition (Simonyan and Zisserman, 2014). Although such tasks are intuitively solved by humans, they originally proved to be the true challenge to artificial intelligence.

Despite their success, when compared with other machine learning techniques, DNNs require more computations due to the deep architecture of the model. Furthermore, developer's ambition for better performance tends to increase the size of the network, leading to longer training times as well as a larger number of computational resources needed for implementation. Currently, researches and practitioners rely on the use of high performance servers to practically implement large scale DNNs. However, such high performance computing clusters incur high power consumption and a large hardware cost, thereby limiting their suitability for low-cost applications such as embedded and wearable IoT devices that require low power consumption and small hardware footprint (Ren *et al*., 2017). These applications increasingly utilise machine learning algorithms to perform fundamental tasks such as natural language processing, speech to text transcription as well as image and video recognition (LeCun *et al*., 2015). Hence, to implement such compute-intensive models in resource constraint systems an alternative implementation needs to be found. In some cases, specialised hardware has been designed using Field Programmable Gate Arrays (FPGAs) and Application Specific Integraded Circuits (ASICs). Nevertheless, there still exists a margin of improvement if the inherent properties and structure of DNNs are further exploited.

This study considers Stochastic Computing (SC) as a low-cost alternative to conventional binary computing. This computing paradigm operates on random bit-streams, where the signal value is encoded by the probability of an arbitrary bit in the sequence being one. Such a representation is particularly attractive as it enables very low-cost implementations of arithmetic operations using simple logic circuits (Alaghi and Hayes, 2013). For example, multiplication and addition can be performed using an AND gate and a Multiplexer (MUX) respectively. Stochastic computing offers very low computation hardware area, high degree of error tolerance and the capability to trade-off computation time and accuracy without any hardware changes (Brown and Card, 2001). It therefore has the potential to implement DNNs with significantly reduced hardware footprint and low power consumption. On the other hand, SC has several disadvantages including accuracy issues due to the inherent variance in estimating the probability represented by the stochastic sequence. Furthermore, an increase in the precision of a stochastic computation requires an exponential increase in the length of the bit-stream (Alaghi and Hayes, 2013), thereby increasing the overall computation time. In general, stochastic arithmetic will be more suitable for an application where the accuracy requirements in the individual computations are relatively low.

FPGAs are a very good accelerator of DNNs. It comprises of integrated chip that allows gate-level reconfiguration of hardware on field. It contains a huge number of logic elements also known as Look Up Tables (LUTs) which can be reconfigured or programmed according to the custom application requirements. Some of the advantages of implementing DNNs on FPGAs are highlighted below:

- FPGA’s have parallel-processing capability that can be used to exploit DNN architecture to inert parallelism, accelerating the DNN inference on embedded devices
- Reconfigurability is the major feature of FPGA that allows specially designed hardware accelerator synthesis for each model, allowing higher optimisation in terms of resource usage and good flexibility for custom user applications
- FPGAs are capable of providing high throughput with low power consumption than existing hardware platforms (Ma *et al*., 2019). Power consumption is very important for embedded systems that have limited area and power supply (such as mobile phones and automotive applications)

The main contributions of our proposed work are:

- A Novel implementation of SC-based DNN on PYNQ Z2 FPGA
- The formulation and implementation of saturation arithmetic in SC
- DNN training using stochastic arithmetic and modified neuron architecture is used
- A scaling scheme is used for DNN inference in SC and an optimization-based scaling scheme is used to learn optimal saturation levels during training

The remainder of this study is structured as follows. Section 1 presents the related work. Section 2 gives fundamental principles of stochastic computing. The number of proposed stochastic processing elements employed in DNNs are presented in section 3. Further, section 4 proposes design and implementation of neural network inference in stochastic computing. Section 5 gives the implementation details and the experimental results. Finally, conclusion and future work are given in section 6.

Related Work

Deep learning principles have been known for many years. However, it wasn’t until the start of the 21st century were advances in hardware technology enabled the development of capable deep learning models. Even today, the training of large scale DNNs is often constraint by the available computational resources.

CPU platforms are in general unable to provide enough computation capacity for training large scale neural networks. Nowadays, GPU platforms are the default choice for neural network training due to the high computation capacity and easy to use development frameworks (Guo *et al*., 2017; Jia *et al*., 2014). Krizhevsky *et al*. (2012) and Facebook AI Group (Yadan *et al*., 2013) train AlexNet, a Convolutional Neural Network (CNN), on multiple GPUs. Li *et al*. (2016) study the memory efficiency of various CNN layers and reveal the performance implication from both data layouts and memory access patterns. Finally, (László *et al*., 2012; Potluri *et al*., 2012) present a GPU based implementation of a cellular neural network, a locally connected recurrent neural network which is widely used in image processing applications.

FPGA based neural network acceleration is an emerging research topic as well. FPGAs can implement high parallelism and potentially surpass GPU in speed and energy efficiency (Guo *et al*., 2017). A main challenge in FPGA based acceleration design is the lack of development frameworks such as TensorFlow and Caffe. To aid the development of deep learning models on FPGAs, (Venieris and Bouganis, 2016) propose a framework for mapping CNNs on FPGAs. Furthermore, the authors (Ma *et al*., 2019; Nakahara *et al*., 2019) propose an FPGA base accelerator to leverage the sources of parallelism in order to achieve an efficient implementation of a deep convolutional neural network. Finally, (Zhu *et al.*, 2020) presents a reconfigurable framework for training CNNs. While viable, the FPGA and GPU based implementations still exhibit a large margin of improvement, mainly because these are general purpose computing devices not specifically optimized for executing DNNs. In addition to such acceleration techniques, DNNs can significantly benefit from the SC technology which allows implementation of complex functions with very simple logic. Stochastic computing has the potential to implement DNNs with significantly reduced hardware footprint when compared to a fixed or floating-point implementation. There have been prior attempts to implement ANNs using stochastic computing.

Qiu *et al*. (2016) proposed a pre-trained deep neural networks implementation on FPGA from VGG (Simonyan and Zisserman, 2014). The 48-bit data representation with dynamic quantization and vector decomposition is used to reduce the size of network, which gave smaller coefficients values that had to be fed to external memory. Zhang *et al*. (2015) used optimization techniques such as transformation and loop tiling to quantitatively analyze computing memory bandwidth and throughput for various DNNs. This representation allowed DNN implementation to achieve high performance of 61.61 GFLOPS. Related work is shown by (Han *et al*., 2016), which results into reduced power consumption by reducing number of weights. More higher level of optimization is proposed by (Dua *et al*., 2020), which uses the OpenGL compiler for DNNs, such as VGG and AlexNet. Hah *et al*. (2019) suggested framework for automatic conversion of deep neural network models into intermediate format (HLS) and then subsequent FPGA implementation.

Qiu *et al*. (2016) utilise SC to implement a Radial Basis Function (RBF) neural network significantly reducing the required hardware. However, RBF neural networks are no longer widely used in deep learning applications as the RBF unit saturates to zero for most of its inputs, making gradient-based optimization challenging. Yu *et al*. (2020) presents a neuron design in SC for DNNs and exploits the energy-accuracy trade-off. Reconfigurable large scale deep learning systems based on SC were designed by (Ren *et al*., 2017). Furthermore, (Li *et al*., 2017; Ren *et al*., 2016) present stochastic computing hardware designs for the implementation of CNNs. In (Ren *et al*., 2016) work, focus is given on weight storage schemes and optimization techniques to reduce area and power consumption of weight storage in hardware. On the other hand, (Li *et al*., 2017) proposes a structure optimization method for a general CNN architecture aiming to minimize area and power consumption while maintaining adequate network accuracy.

In summary, the aforementioned works have proposed certain neuron designs using SC in order to satisfy the computing limitations in resource-constraint applications such as embedded systems. However, they only consider the implementation of neural network inference using SC hardware. Moreover, only a certain activation, namely the hyperbolic function is considered whose usage has reduced significantly since the introduction of the rectified linear unit. Despite previous work, there still lacks a detailed investigation regarding the scaling scheme used to implement a neural network using SC hardware. Finally, there is no existing work that investigates comprehensively how stochastic computing can be incorporated during the training stage of a DNN and how this can affect the performance of the neural network on the recognition task.

First, a detailed investigation of the stochastic processing elements employed in DNNs is conducted. Amongst them, a stochastic approximation of the max function is presented and subsequently used to approximate the rectified linear unit in SC. As addition in SC is performed in a scaled manner, saturation arithmetic architectures are proposed to alleviate large down-scaling parameters that undermine precision in the computations. Combining several building blocks, a scheme is proposed for the implementation of neural network inference in SC. Finally, a modified neuron architecture is used for training DNNs which are SC compatible and can be implemented efficiently using SC hardware. Experimental results using the MNIST dataset demonstrate that the proposed inference scheme can implement a neural network in SC without increasing the error by more than 2.34%.

Stochastic Computing

Stochastic computing relies on probability theory, where a probability number is represented by a bit-stream of chosen length and its value is determined by the probability of an arbitrary bit in the bit-stream being one (Gaines, 1969). For example, a stochastic bit-stream containing 75% of ones and 25% of zeros represents the number $p = 0.75$, reflecting the fact that the probability of observing a one in an arbitrary bit position is 0.75. Clearly, when compared to a binary radix representation, the stochastic representation is not very compact. However, it leads to very low-complexity arithmetic units which was a primary concern in the past. For example, multiplication in stochastic arithmetic can be performed by a single AND gate. Consider two input stochastic streams that are logically ANDed and assume that the probability of observing a one in each stream is $p1$ and $p2$ respectively. Then, assuming that the inputs are suitably uncorrelated or independent, the probability of any bit in the output of the AND gate being a one is $p1 \times p2$. Figure 1a shows this operation.

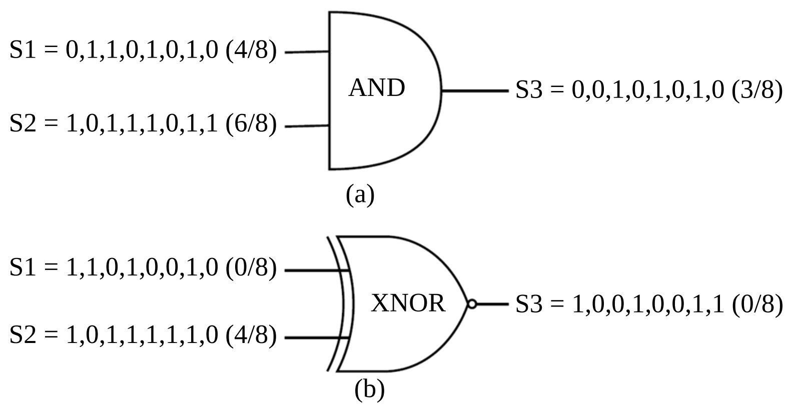

S1 = 0,1,1,0,1,0,1,0 (4/8)
S2 = 1,0,1,1,1,0,1,1 (6/8)
AND
S3 = 0,0,1,0,1,0,1,0 (3/8)
(a)
S1 = 1,1,0,1,0,0,1,0 (0/8)
S2 = 1,0,1,1,1,1,1,0 (4/8)
XNOR
S3 = 1,0,0,1,0,0,1,1 (0/8)
(b)

Fig. 1: Stochastic Multiplication using (a) AND gate in unipolar format and (b) XNOR gate in bipolar format

The multiplication operation is a closed operation on the interval [0, 1] or [-1, 1] for unipolar and bipolar signals respectively. In the bipolar representation an XNOR gate performs multiplication between two Bernoulli sequences Fig. 1b and 2a. The XNOR output is logic 1 whenever the two inputs are either both logic 0 or logic 1. Denoting by *S*1 and *S*2 the inputs to the XNOR gate and *S*3 the output, then for the bipolar representation one has:

$$\begin{aligned} P_{S3} &= \left(P_{S1} \times P_{S2}\right) + \left(P_{\bar{S}1} \times P_{\bar{S}2}\right) \\ &= \left(P_{S1} \times P_{S2}\right) + (1 - P_{S1}) \times (1 - P_{S2}) \\ &= 2P_{S1} \times P_{S2} - P_{S1} - P_{S2} + 1 \end{aligned}$$

Using the fact that $P_{S1} = \frac{s1+1}{2}$ and $P_{S2} = \frac{s2+1}{2}$, then:

$$P_{S3} = \frac{s1s2+1}{2}$$

For bipolar signals, $s3 = 2P_{S3}-1$ therefore:

$$s3 = 2\left(\frac{s1s2+1}{2}\right) - 1 = s1 \times s3 \tag{1}$$

The stochastic multiplier gives an estimate of the result and if *S*1 and *S*2 are independent Bernoulli bits then output *S*3 is also a Bernoulli sequence. If there is no error in the approximation, the final value of *s*3 might not be equal to the product *s*1 × *s*2. Factors such as fluctuation in bit stream and quantisation error could be the reason behind SC based representation error. In this study, we have implemented stochastic multiplier in target programming language.

Proposed Stochastic Processing Elements

This section analyses a number of stochastic processing units employed in deep neural networks. This study considers both combinational logic and sequential circuit for processing stochastic bit-streams. Without loss of generality, the bipolar format of the stochastic processing units is mainly considered.

Addition

Addition and subtraction in stochastic arithmetic are slightly more complex operations than multiplication. This is due to the fact that addition and subtraction are not closed operations on the interval [0, 1] or [-1, 1]. The result of adding two numbers that lie within [-1, 1] does not necessarily lie within [-1, 1]. For this reason, a scaled add operation is used in SC in order to map the output of the adder from [-2, 2] to [-1, 1]. The weighted sum of two probabilities, $\alpha p1+(1-\alpha)p2$; where $0 \le \alpha \le 1$, lies within [-1, 1] and is representable in the stochastic computing domain. Such a computation can be realised using a two-input multiplexer where the select line is driven by the selecting probability α (Gaines, 1969). Consider the MUX in Fig. 2b. The probability of a logic one appearing at the output is equal to:

$$P_y = P_A \cdot P_S + P_B \cdot P_S \tag{2}$$

By choosing $P_S = 0.5$, for bipolar signals one has:

$$y = 2P_Y - 1 = \frac{a+b}{2} \tag{3}$$

In other words, the MUX generates an output with a generating probability that is the weighted sum of the input probabilities. $y = a \otimes b = \frac{(a+b)}{2}$ will be used to denote the scaled addition operation in stochastic computing. Note that for bipolar signals $P_S = 0.5$ corresponds to a stochastic bit-stream having 50% zeros and 50% ones. Scaled subtraction can be implemented using the same MUX unit simply by inverting the input to be subtracted. The scaled adder is implemented in the target environment and its operation is verified empirically.

Squaring

The squaring operation is very similar to that of multiplication. However, attempting to square a stochastic signal by connecting it to both inputs of a XNOR gate results in a sequence that is always logic 1. This is because the two input signals are correlated with each other. This effect can be avoided by multiplying a stochastic sequence with its delayed, by one clock cycle, copy sequence. In that case, the two inputs are uncorrelated and the output sequence will approximate the square value of the input. The delay can be realised in hardware by placing a D-type flip-flop in one of the inputs of the XNOR gate as illustrated in Fig. 2c. Flip-flops used in this context perform no computation. Instead, they are used to statistically isolate two cross-correlated sequences (Gaines, 1969).

Inner Product

The inner product is the core operation of artificial neurons both for feedforward networks but also for convolutional networks. Hence, to effectively implement neural networks using stochastic computing an efficient stochastic inner product unit is required. Similar to addition and subtraction, the inner product is not a closed operation on the interval [-1, 1], hence a scaled inner product is utilised in the context of SC. As proposed by (Gaines, 1969), the two-input scaled adder can be extended to the weighted sum of an arbitrary number of input signals using the same MUX architecture. For a MUX unit with *N* inputs, this is done by selecting one of the input lines at random, with a certain probability of selecting each one and connecting the selected input line to the output line for a single clock cycle, i.e., for a single bit.

One way to implement the inner product would be to use XNOR multiply units to compute the products $w_i x_i$ for all *i* followed by an equally weighted N-input MUX unit to accumulate the results. In contrast to the implementation above, this approach requires to convert weight values into stochastic bit-streams. As the implementation given by Algorithm 1 provides greater flexibility in software, it is preferred in the context of this study.

Algorithm 1 Stochastic Inner Product

```
Input: Number of inputs N
Floating point weight coefficient $w_i \in R$
Stochastic bit-stream $X_i \in R^L$ representing $x_i \in [-1, 1]$
Output: Bit-stream $Y \in R^L$
Integer $s_{out} \in R$
for $i = 0$ to $N-1$ do
    Invert inputs if $w_i < 0$ then
        $Z_i = X_i$
    else
        $Z_i = X_i$
    end
    Define probability distribution;
    $\alpha_i = \frac{|w_i|}{\sum_j |w_j|}$
end
Generate Output bit-stream;
  for $j = 0$ to $L-1$ do
    $i$ = Draw a sample according to $\alpha_i$ $Y[j] = Z_i[j]$ $S_{out} = \sum_{i=0}^{N-1} |w_i|$
end
```

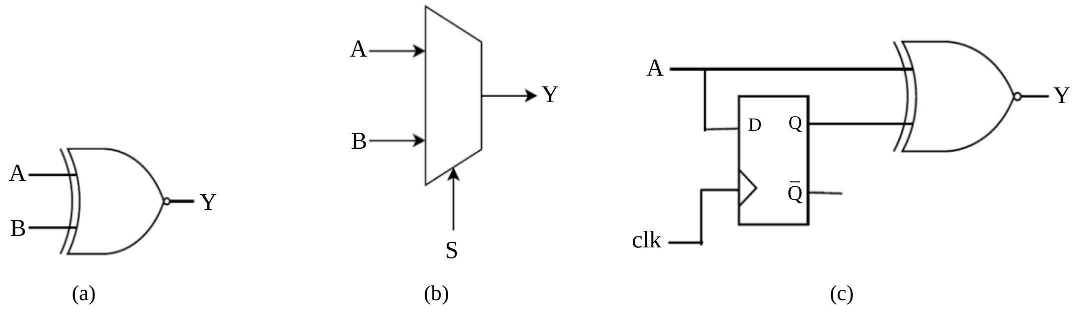


Fig. 2: Stochastic Units; (a) Bipolar Multiplier; (b) Scaled Adder; (c) Squaring Circuit

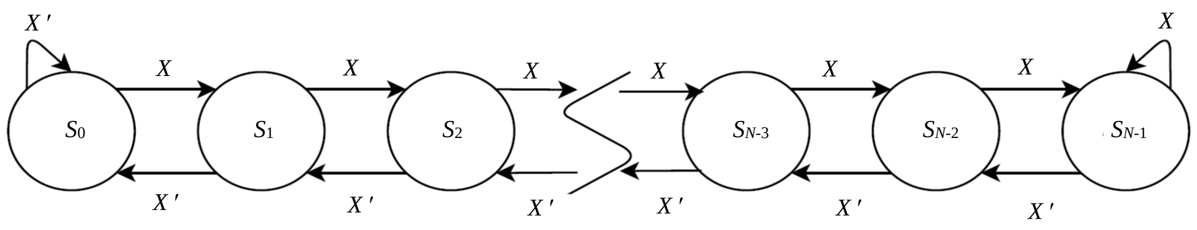


Fig. 3: Linear finite state machine

Proposed FSM Based Elements in SC

Deep neural networks employ highly non-linear activation and output units such as the hyperbolic tangent function or the rectified linear unit. The implementation of such non-linear functions with combinational logic is sometimes impossible and is in general not straightforward so we have used FSM based elements for SC based DNN.

The basic form of the proposed FSM is illustrated in Fig. 3. It consists of a set of N states arranged in a linear form (i.e., a saturating counter). Usually, $N = 2^K$ is chosen where K is a positive integer. This is a no skips model. That is transitioning from the first to the last state must occur through a set of transitions through all of the intermediate states (Gaines, 1969). Additionally, the state transitions are controlled by the input stochastic sequence X which is assumed to be a Bernoulli sequence and the output Y at each clock cycle is determined entirely by the current state. Note that the states S_0 and $S_{N\text{-}1}$ have saturating effects.

Stochastic Maximum Function

Undoubtedly, a stochastic implementation of a max function is of particular importance for the purpose of implementing modern deep neural networks in stochastic computing. However, in contrast to a conventional radix-2 representation, where individual bits are weighted by their position in the digit-vector, in stochastic arithmetic all the bits in the stochastic sequence are equally weighted. Thus, neither the value nor the sign of the stochastic signal is related to the exact position of the ones and zeros in the bit-stream. Instead, the ratio of ones to the length of the bit-stream determines both the sign and value of the signal. Thereby, a processing element that computes the maximum (or minimum) between two stochastic signals cannot rely on the position of the individual bits in the input bit-streams, as a binary equivalent could do.

An approximation of the max function in stochastic arithmetic, namely Smax, with both input and output signals encoded as bipolar stochastic bit-streams may be implemented using the configuration shown in Fig. 4. The basic idea of the stochastic max unit is to compute the difference between the inputs A and B, i.e., A-B and based on that to generate a select line signal that will choose the maximum between the two inputs. Based on the architecture in Fig. 4, the difference is computed using the leftmost MUX unit (in combination with the NOT gate). The resulting bit-stream is fed into the *Stanh* unit which is implemented using the FSM. Thus, if P_A is larger than P_B, then Stanh tends to stay on the high state side, whereas if P_B is larger than P_A, then Stanh tends to stay on the low state side. Finally, the rightmost MUX unit in Fig. 4 selects A if the *Stanh* output is at the high state and B if the *Stanh* output is at the low state. Thus, the output of the circuit shown in Fig. 4 is equal to max(A,B). A stochastic minimum function can be achieved by simply permuting the inputs of the output (i.e., rightmost) MUX unit in Fig. 4. The stochastic max unit is implemented in the target language.

Proposed Saturation Arithmetic in Stochastic Computing

Accumulation in stochastic arithmetic needs to be performed in a scaled manner as addition and subtraction are not closed operations on the interval [-1, 1]. The output of a two-input stochastic adder is therefore scaled by 1/2. Cascading N such scaled adders, results in an output that is down-scaled by 2^N. Such a down-scaling phenomenon can cause severe accuracy loss in the overall computation especially when N is large, as a stochastic computing system using word lengths of size 2^L can only represent values as low as $1 = 2^L$, which may be insufficient to represent the down-scaled output when N is large. This becomes even worst if the values involved in the computation are themselves small. Note that such imprecision cannot be compensated by post-processing (i.e., up-scaling) the output of the overall system as the information is already lost during the down-scaling procedure in the stochastic domain. Similarly, the corresponding output of the stochastic inner product is down-scaled by $\sum_{i=1}^{N}|w_i|$, which can be significantly large when either N or $|w_i|$ are large.

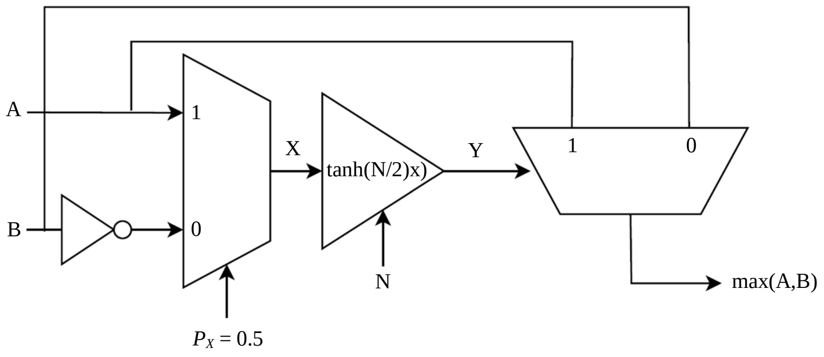


Fig. 4: Stochastic max unit

The objective while designing saturation arithmetic units in the stochastic domain is to mitigate the down-scaling effect by worst-case scalings at the output of a stochastic accumulator, thus increase precision while minimizing possible representation errors in the stochastic encoding of the result. Note that in contrast to the fixed-point realization of a saturation system in conventional binary computing, in stochastic computing truncation of low order bits is not possible as all bits in a stochastic sequence are equally weighted. Hence, only saturations can be considered as a possible way of realizing saturation arithmetic in SC.

Stochastic Computing Based Neural Network

This section addresses the design and implementation of neural network inference in stochastic computing. Without loss of generality, emphasis is given on feed forward neural networks, i.e., multi-layer perceptrons.

Similar to input data values, encoding the model's parameters by means of stochastic bit-streams without any saturation error requires that the trained coefficients lie inside the range [-1, 1], either inherently or after appropriate processing. In contrast however to the primary input values, weights and biases can be scaled individually so that each coefficient can be represented without compression error in stochastic computing. This is because these are trained coefficients and are fixed during inference. Hence, once the scaling of each coefficient is determined it will not change for any input data point.

Following the design and analysis of stochastic processing elements in section 3, the implementation of an inner product in stochastic computing according to algorithm 2 does not require to convert the weight values into stochastic bit-streams. Instead, their absolute values are used to define a selecting probability distribution over the inputs of the MUX unit. On the other hand, the stochastic adder requires that both inputs are encoded as stochastic bit-streams. Therefore, in terms of simulating the neural network inference within a software environment, only the trained biases need to be converted into stochastic bit-streams, whereas the learned weights can be kept in their floating-point representation. To convert bias coefficients to stochastic bit-streams, each bias term b can be down-scaled by:

$$s_{bias} = 2^{\log_2 |b|} \tag{4}$$

Then the down-scaled coefficient, $b' = \frac{b}{s_{bias}}$, can be represented without any compression error in stochastic computing.

Network Scaling

In the context of inference, once the scaling factor at the input is fixed, the scaling coefficient of every node in the SC equivalent graph can be determined. This is exactly due to the fact that during inference the model's parameters are fixed and known. The process of specifying the scaling of every node in the network is termed scaling scheme and is based on the scaling parameter at the output of every individual processing element that is employed in the SC network graph. The scheme proposed in this study is based on forward propagation of known information on data ranges through the network data flow graph. The process is described in the remaining of this section considering feedforward network data flow graphs.

Algorithm 2 Input Product Scaling Scheme

```
Input: Number of inputs $m \in R$
Weight coefficient $w \in R^m$
Input Scaling Factor $s \in R^m$
Output: Input re-scaling coefficients $r_{in} \in R^m$
Output re-scaling coefficient $r_{out} \in R$
Output scaling factor $s_{dot} \in R$
$s_{\max} = \max\{s_0, s_1, \ldots, s_{m-1}\}$
  for i = 0 to m-1 do
      $r_{ini} = \frac{s_i}{s_{\max}}$
      $w_{sum} = \sum_{i=0}^{m-1} |w_i|$
      $s_{dot} = 2^{\lceil \log_2 (s_{in} \times w_{sum}) \rceil}$
      $r_{out} = \frac{s_{\max} \times w_{sum}}{s_{dot}}$
end
```

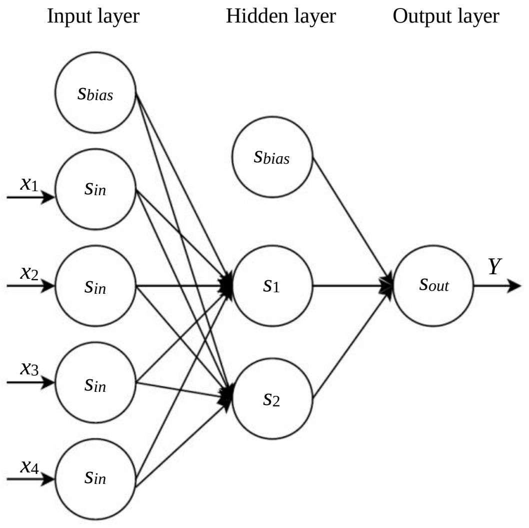


Fig. 5: Deep neural data flow graph

As an illustrative example, consider the network graph shown in Fig. 5. The network has four inputs, $x_1,\ldots,x_4$, a single hidden layer with two units and a single output y. Next, consider a single data point $x \in R^4$ and assume the weight matrices in the hidden and output layer are given by $W^{(1)} \in R^{4\times2}$ and $W^{(2)} \in R^{2\times1}$ respectively. Furthermore, the biases are given by $b^{(1)} \in R^2$ and $b^{(2)} \in R$ for the hidden and output layers respectively. The activations in the hidden layer are therefore calculated as:

$$h^{(1)} = \phi\left(W^{(1)T}\ x + b^{(1)}\right) \tag{5}$$

and the output of the network is given by:

$$y = \phi\left(W^{(2)Th^{(1)}} + b^{(2)}\right) \tag{6}$$

where, ϕ is the activation function. To start with, assuming that the input data values x_i lie in some interval [$-l$, u], the input scaling factor s_{in} is selected. Thus, every input x_i is scaled by s_{in} and the down-scaled inputs $x'i = x_i/s_in$ are converted into stochastic bit-streams. Next, consider the first activation $h_1^{(1)}$ in the hidden layer. This is computed as follows:

$$h_1^{(1)} = \phi\left(\sum_{i=0}^{3} w_{i,1}^{(1)} \cdot x_i\ + b_1\right) \tag{7}$$

Algorithm 3 Bias addition scaling scheme

```
Input: Bias Scaling Factor $s_{bias} \in R$
Input scaling factor $s_{dot} \in R$
Output: Input re-scaling coefficients $r_{bias}, r_{dot} \in R_m$
Output scaling factor $s_{out} \in R$
$s_{\max} = \max\{s_{bias}, s_{dot}\}$
  $r_{out} = \frac{s_{dot}}{s_{\max}}$
  $r_{bias} = \frac{s_{bias}}{s_{\max}}$
  $s_{out} = 2\times s_{\max}$
```

The inner product between the weights and the down-scaled input values can be calculated in stochastic computing using the implementation given in algorithm 2. The output will be a stochastic bit-stream representing the down-scaled weighted sum. Recall that the output of algorithm 2 is associated to a scaling coefficient $w_{sum} = \sum_{i=0}^{3}\left|w_{i,1}^{(1)}\right|$. Hence, the inner product between $w_{i,1}^{(1)}$ and $\{x_i\}$ in 7, when computed in stochastic computing will be down-scaled by a factor of $s_{in} \times w_{sum}$. Note that $s_{in} \times w_{sum}$ is fixed for all data points and can be computed in advance as the matrix $W^{(1)}$ is known. Finally, to maintain consistency throughout the network structure, the output of the stochastic inner product is re-scaled accordingly so that the scaling factor associated with it is given by:

$$s_{dot} = 2^{\left\lceil \log_2\left(s_{in}\times w_{sum}\right)\right\rceil} \tag{8}$$

that is, the next integer power of 2 of $s_{in} \times w_{sum}$. Since $s_{dot} \geq s_{in} \times w_{sum}$, the aforementioned re-scaling of the inner product output can be realised by means of a XNOR multiplier with inputs the inner product output and a bit-stream representing $\frac{s_{in} \times w_{sum}}{sdot} \leq 1$. This is easy to implement in hardware and does not incur significant hardware or delay overheads.

The proposed scaling scheme easily extended to feedforward networks of arbitrary depth and width. In summary, the proposed scheme consists of two main procedures. The one associated with the MUX-based inner product between weights and features and the one associated with the addition of the bias term. These are summarised, in their most general form, in algorithms 2 and 3. The re-scaling coefficients are used at the input and output of stochastic accumulators to appropriately re-scale the corresponding bit-streams using XNOR multipliers. Algorithm 4 illustrates a possible implementation of the Stanh FSM architecture.

Effectively, the scaling coefficients at every node of the network specify the range of values that the corresponding signal can take and are the same for all data points. This is because both the input scaling parameter selected but also the scaling at the output of each individual processing unit is worst-case scalings. Hence, the procedures given in algorithms 2 and 3 need to be executed only during the construction of the SC network graph and the re-scaling multipliers need to be inserted wherever is needed. Once these actions are done, the SC based neural network graph is completed and remains fixed during inference run time. As briefly discussed this is a desired consequence as it allows the hardware infrastructure to remain fixed during run time.

Finally, in the event where saturation arithmetic is employed, appropriate saturation levels need to be determined. A standard approach when creating a saturation arithmetic implementation, is to determine saturation levels through simulation. For the purpose of neural network inference, this is described as follows. Test data are applied to the network and the peak value reached by each signal is recorded. Internal scalings and thereby saturation levels, are then selected to ensure that the full dynamic range afforded by the signal representation would be used under excitation with the given input vectors (Constantinides *et al*., 2003).

Algorithm 4 Stochastic approximation of the hyperbolic tangent function

```
Input: Bit-stream $X$
Number of states $N$
Number of stochastic samples $L$
Output: Number of stochastic samples $L$
Initialize FSM Parameters
    $S_{\min} = 0$
    $S_{\max} = N-1$
    $S_{bound} = \frac{N}{2}$
    Initialize state sequence
    $S = s_{bound}$
    for $i = 0$ to $L-1$ do
        $S$
end
tate transition
    if $X[i] == 0$ then
        $S = S-1$
else
        $S = S+1$
end
Saturate the counter
    if $S < S_{\min}$ then
        $S = S_{\min}$
else
        $S = S_{\max}$
end
Output logic
    if $S \geq S_{bound}$ then
        $Y[i] = 1$
else
        $Y[i] = 0$
end
```

Once the output signals are computed, conversion from stochastic arithmetic to floating-point arithmetic can be done effectively. Each conversion outcome will be intrinsically down-scaled by the scaling associated to the corresponding bit-stream. Thus, the outcome of the conversion must be up-scaled, in floating-point, by an amount equal to the scaling coefficient of the signal.

Experimental Results

We have prototyped a DNN accelerator on an FPGA board. The main goal of this is to validate our SC-based DNN accelerator as well as to assess our SC algorithm's suitability for FPGA implementations. We have used a XILINX ZYNQ XC7Z020 board, which includes PYNQ Z2 FPGA and a DDR3 memory. Figure 6 illustrates the system architecture, where the Jupyter Notebook is used to run the software part and an AXI bus and an AXI memory controller are used to connect hardware modules to the DDR3 memory. The main module is designed in Verilog HDL, for the conventional binary representation as well as proposed SC. For the prototyping we have used DNN, with network size given in Table 1. All the layers are implemented in software on the Python as discussed above. We have verified the correct operation, matching with the software simulation results produced by Tensorflow on Google colab GPU. The used development board (PYNQ Z2 board) prototype incorporates a low-cost FPGA (ZYNQ XC7Z020-1CLG400C) where the DNN circuit is configured. However, we also incorporate an UART and a state's machine in the FPGA to allow writing the DNN inputs and reading the DNN outputs from the PC.

Table 1 shows our FPGA synthesis result, after respective mappings. LUT-based MACs are used for the binary case as DSP blocks are generally not used in SC-based designs. Xilinx RTL synthesis tool based optimization is considered for design. From the Table 1 it is very clear that SC-based DNN use much less resources compared with the conventional binary design. Thus SC-based DNN can be useful in low cost applications. Our proposed SC based DNN attains near 1 cycle latency for MNIST, thus successfully achieve the higher efficiency in terms of area and power.

Table 1: Comparison of floating point DNN and SC based DNN implementations on FPGA @ 100 MHz Frequency with 728-128-10 network size

Dataset	Models	LUT	FF	DSP	Freq.
MNIST	Binary radix	46172	45931	NA	100 (MHz)
	Proposed	40917	46612		

Table 2: Hardware complexity of the proposed FSM based processing element @ 100 MHz in FPGA Prototype

Activation Function	1024		512		256		128	
	Area (mm^2)	Power (W)	Area (mm^2)	Power (W)	Area (mm^2)	Power (W)	Area (mm^2)	Power (W)
tanh(s)	0.029	3.9e-6	0.069	9.6e-6	0.119	1.93e-6	0.17	2.6e-6
ReLU(2s)	0.034	4.1e-6	0.086	1.1e-5	0.150	2.1e-6	0.21	2.94e-6
exp(-s)	NA	NA	0.453	6.81e-5	0.481	7.89e-5	0.51	9e-5

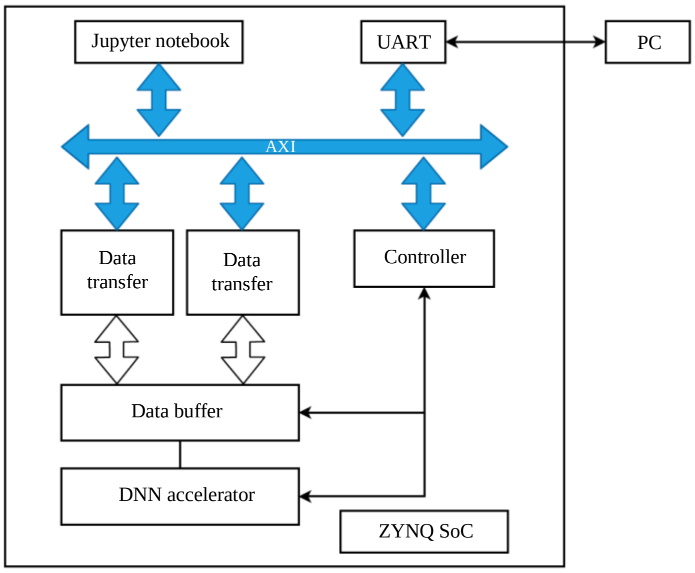
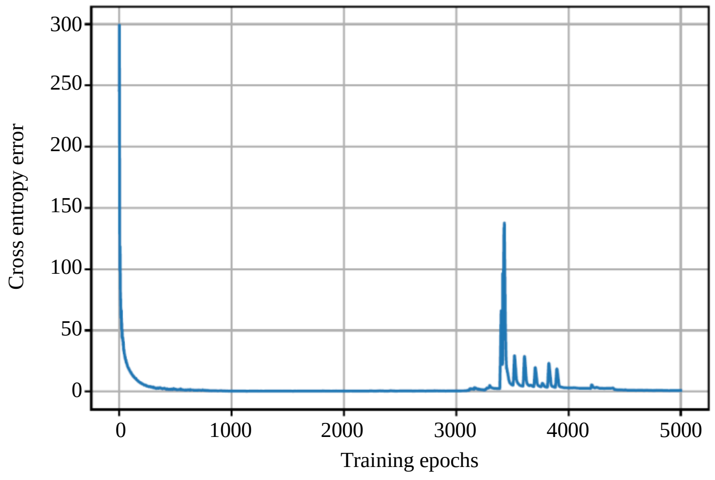
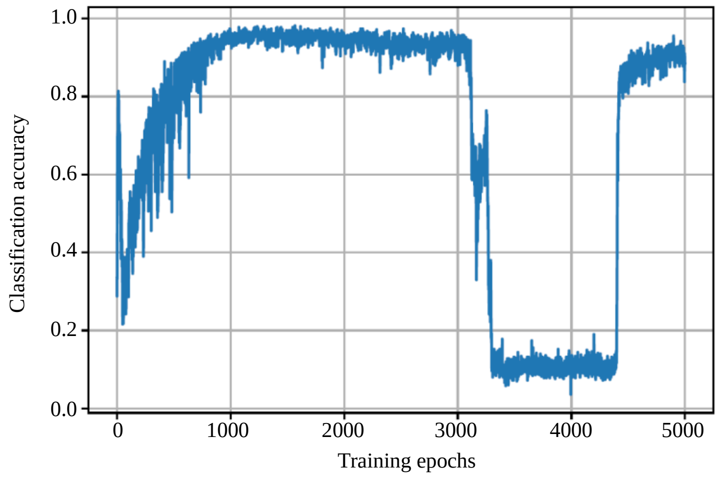


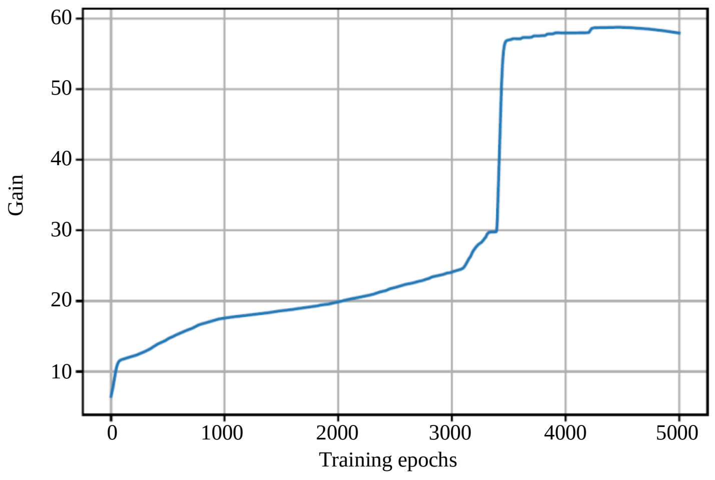
Fig. 6: Block diagram of SC-based DNN accelerator implementation on ZYNQ XC7Z020



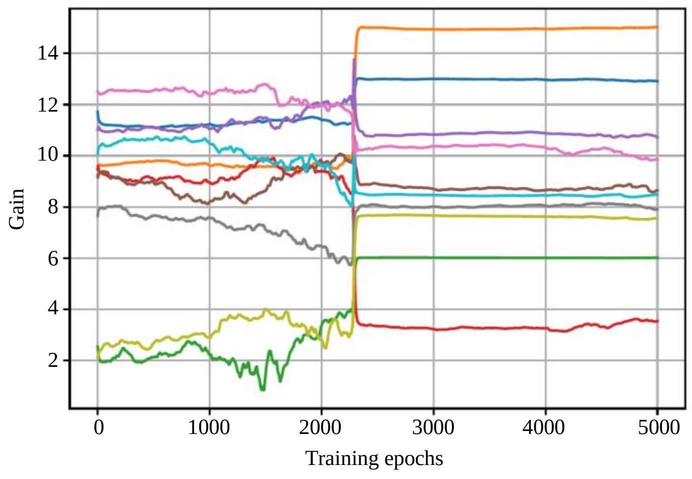
(a)



(b)



(c)



(d)

Fig. 7: Results from the training process of SC based DNN; (a) Training loss versus epochs; (b) Training accuracy versus epochs; (c) Gain parameters in the hidden layer; (d) Gain parameters in the output layer

The hardware complexity of the proposed FSM-based processing elements @ 100 MHz in FPGA is also summarized in Table 2. The implementation results show that the proposed processing elements consumes approximately 6× more power at most while having 7× times less latency, which gives us lower power consumption, compared to the conventional elements (i.e., FSM-based elements with 1024 bit). Which denotes the latency. The trained network with SC achieves a classification accuracy of 98.17% on the training set and 97.76% on the test set as compared to the trained model using conventional floating point, achieving classification accuracy of 87.05% on the training set and 87.03% on the testing set. Results for this network are collectively shown in Fig. 7. The maximum throughput that can be achieved by SC based accelerator design is 1.877 GOP/s.

As per experiment, it was observed that more training epochs is required for the DNN architecture to achieve an acceptable accuracy on the training and testing sets. Thus, the number of epochs is increased to 5000 and a batch size of 500 is used in each epoch to compute the gradients and update the model parameters. Furthermore, long stochastic bit-stream length have been used in implementation so that the additive Gaussian noise becomes zero. Perhaps unsurprisingly, the behaviour observed in the preceded experiment is noticed in this one as well. That is, sudden increments in the gain coefficients, which control the saturation levels of the SC neuron, cause a sharp increase in the loss function during training. Once again, the network seems to appropriately adapt the remaining trainable parameters to the updated gain coefficients in order to ensure that the loss is minimized. As Fig. 8 suggests, there exist some data that will cause a large saturation error once signal values are clipped to ±1. Nonetheless, as previously argued, it seems that the network consciously takes the decision to increase the gain coefficients (and thereby decrease the scaling factors), to avoid loss of precision due to large down-scale parameters in the network graph. During the training, weights and biases are continuously updated by the learning algorithm thus the scaling coefficients of the computational graph will, in general, vary during run-time and compared to accuracy it is shown in Fig. 9. In a sense, the network prefers to increase precision in the computations at the cost of a non-zero saturation error for some data points. A reasonable interpretation is that such data points occur rarely, thus it is preferable not to precisely accommodate computations related to those points and instead reduce the scaling coefficients to facilitate computations for the remaining data points with higher precision.

Comparison with Existing SC-DNNs

In this section we do a quantitative comparison with existing SC based DNNs operating on unipolar and bipolar representations, which is employed by many existing DNNs.

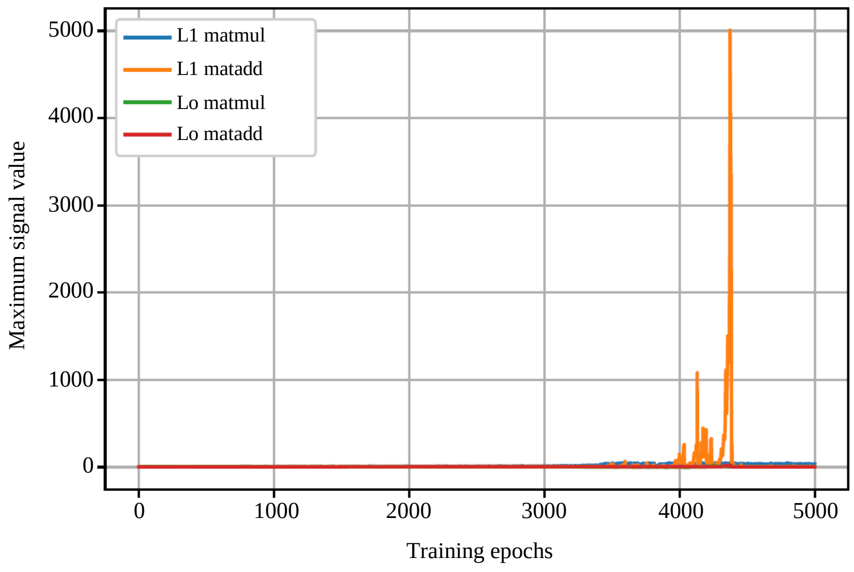


Fig. 8: Max signal value from each saturated operation prior clipping to ±1

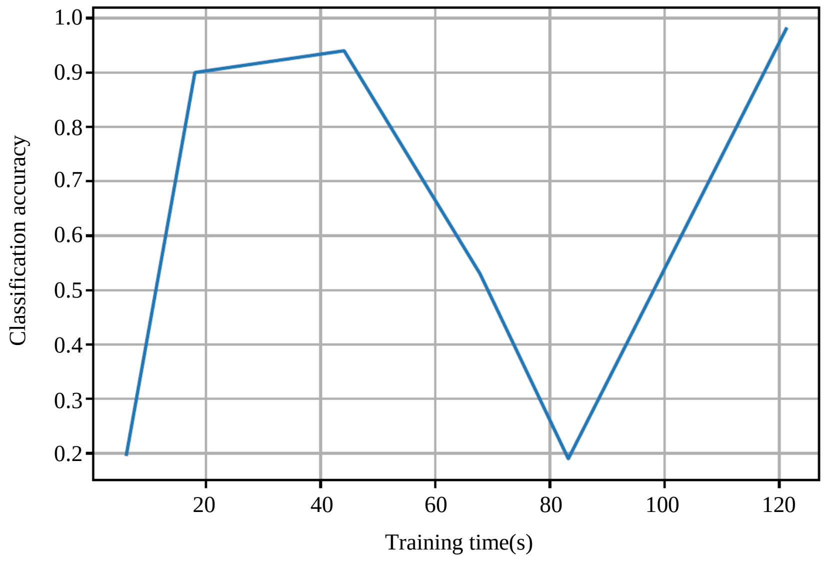


Fig. 9: Training time versus classification accuracy

Table 3: Existing hardware platforms Vs proposed approach

	Dataset	Network type	Year	Platform type	Throughput (Images/sec)	Area (mm^2)	Power (W)	Accuracy (%)
EIE-64PE Han *et al*. (2016)	ImageNet	CNN	2016	ASIC	81967	40.8	0.59	NA
ArXiv'17 Ren *et al*. (2017)	MNIST	CNN	2017	ASIC	781250	36.4	3.53	98.26
DAC'17 Sim and Lee (2017)	MNIST	DNN	2017	FPGA	NA	0.06	0.025	98.00
SIGPLAN'18 Cai *et al*. (2018)	MNIST	BNN	2018	FPGA	321543.4	NA	0.560	97.81
IEEE Trans.'18 Alawad (2018)	ImageNet	CNN	2018	FPGA	NA	NA	3.61	86.77
IEEE ISCAS'19 Lammie and Azghadi (2019)	MNIST	DNN	2019	FPGA	NA	NA	6.80	98.13
Proposed Model	MNIST	DNN	2020	FPGA	NA	0.61	1.89	98.17

Table 3 shows the results of our proposed SC based DNNs together with other implementations. It includes several software and hardware implementations using FPGAs and ASICs. Hardware neural networks such as Spiking Neural Network (SNN) or Bayesian Neural Network (BNN) have been implemented on various platforms. According to Table 3, the proposed SC based DNN is more area efficient: The area of ArXiv'17 (Ren *et al*., 2017) is much more then the area of our proposed SC based DNN. Moreover, our proposed SC based DNN also have outstanding performance in terms of area, power and accuracy.

Conclusion

This study considers stochastic computing, a low-cost alternative to conventional binary computing to implement modern deep neural networks. It was found that the worst case scaling parameters that are inherently introduced by stochastic arithmetic tend to be overly pessimistic, undermining the implementation of neural network inference in SC. It was shown that by appropriately applying saturation arithmetic, the SC network can achieve the higher level of accuracy then the conventional floating-point network. Extending the implementation of neural network inference in stochastic computing, a modified training procedure was proposed aiming to capture the limitations of the stochastic representation within the training phase of the model. Interestingly, it was found that this allows the network to develop its own knowledge regarding both the recognition task as well as the alternative representation that we are trying to impose. The network seems to identify the limitations of stochastic computing and appropriately modifies its parameters to address them. As a consequence, a subsequent implementation of the inference algorithm using SC hardware could benefit significantly by this training procedure. Finally, it was found that the proposed training approach can even improve the network's predictions, both in and out of sample.

Further research can be conducted on several other hardware platforms. It will allow to quantitatively access the overheads introduced by saturation arithmetic in SC as well as to identify if there exists a break-even point. Additional experiments need to be conducted with deeper network architectures as well as testing with alternative datasets can be done.

Funding Information

No Funding has been received for this work.

Author's Contributions

All authors have equally contributed to this work.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and there are no ethical issues involved.

References

Abdel-Hamid, O., Mohamed, A. R., Jiang, H., Deng, L., Penn, G., & Yu, D. (2014). Convolutional neural networks for speech recognition. IEEE/ACM Transactions on audio, speech and language processing, 22(10), 1533-1545.

Alaghi, A., & Hayes, J. P. (2013). Survey of stochastic computing. ACM Transactions on Embedded computing systems (TECS), 12(2s), 1-19.

Alawad, M. (2018). Scalable fpga accelerator for deep convolutional neural networks with stochastic streaming. IEEE Transactions on Multi-Scale Computing Systems, 4(4), 888-899.

Brown, B. D., & Card, H. C. (2001). Stochastic neural computation. I. Computational elements. IEEE Transactions on computers, 50(9), 891-905.

Cai, R., Ren, A., Liu, N., Ding, C., Wang, L., Qian, X., ... & Wang, Y. (2018). VIBNN: Hardware acceleration of Bayesian neural networks. ACM SIGPLAN Notices, 53(2), 476-488.

Constantinides, G. A., Cheung, P. Y., & Luk, W. (2003). Synthesis of saturation arithmetic architectures. ACM Transactions on Design Automation of Electronic Systems (TODAES), 8(3), 334-354.

Deng, L., & Yu, D. (2014). Deep learning: methods and applications. Foundations and trends in signal processing, 7(3–4), 197-387.

Dua, A., Li, Y., & Ren, F. (2020, May). Systolic-CNN: An OpenCL-defined Scalable Run-time-flexible FPGA Accelerator Architecture for Accelerating Convolutional Neural Network Inference in Cloud/Edge Computing. In 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (pp. 231-231). IEEE.

Gaines, B. R. (1969). Stochastic computing systems. In Advances in information systems science (pp. 37-172). Springer, Boston, MA.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning (Vol. 1, No. 2). Cambridge: MIT press.

Guo, K., Zeng, S., Yu, J., Wang, Y., & Yang, H. (2017). A survey of FPGA-based neural network accelerator. arXiv preprint arXiv:1712.08934.

Hah, T. K., Liew, Y. T., & Ong, J. (2019). Low Precision Constant Parameter CNN on FPGA. arXiv preprint arXiv:1901.04969.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., & Dally, W. J. (2016). EIE: efficient inference engine on compressed deep neural network. ACM SIGARCH Computer Architecture News, 44(3), 243-254.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... & Darrell, T. (2014, November). Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia (pp. 675-678).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, (pp. 1097-1105). Red Hook, NY, USA. Curran Associates Inc.

Lammie, C., & Azghadi, M. R. (2019, May). Stochastic computing for low-power and high-speed deep learning on FPGA. In 2019 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). IEEE.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436-444.

Li, C., Yang, Y., Feng, M., Chakradhar, S., & Zhou, H. (2016, November). Optimizing memory efficiency for deep convolutional neural networks on GPUs. In SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 633-644). IEEE.

Li, J., Ren, A., Li, Z., Ding, C., Yuan, B., Qiu, Q., & Wang, Y. (2017, January). Towards acceleration of deep convolutional neural networks using stochastic computing. In 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC) (pp. 115-120). IEEE.

László, E., Szolgay, P., & Nagy, Z. (2012, August). Analysis of a gpu based cnn implementation. In 2012 13th International Workshop on Cellular Nanoscale Networks and their Applications (pp. 1-5). IEEE.

Ma, Y., Cao, Y., Vrudhula, S., & Seo, J. S. (2019). Performance modeling for CNN inference accelerators on FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(4), 843-856.

Nakahara, H., Sada, Y., Shimoda, M., Sayama, K., Jinguji, A., & Sato, S. (2019, September). FPGA-based training accelerator utilizing sparseness of convolutional neural network. In 2019 29th International Conference on Field Programmable Logic and Applications (FPL) (pp. 180-186). IEEE.

Potluri, S., Fasih, A., Vutukuru, L. K., Machot, F. A., & Kyamakya, K. (2012). CNN Based High Performance Computing for Real Time Image Processing on GPU. In: Unger H., Kyamaky K., Kacprzyk J. (eds). Autonomous Systems: Developments and Trends, (pp. 255-266). Springer Berlin Heidelberg, Berlin, Heidelberg.

Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., ... & Wang, Y. (2016, February). Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 26-35).

Ren, A., Li, Z., Ding, C., Qiu, Q., Wang, Y., Li, J., ... & Yuan, B. (2017). Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. ACM SIGPLAN Notices, 52(4), 405-418.

Ren, A., Li, Z., Wang, Y., Qiu, Q., & Yuan, B. (2016, October). Designing reconfigurable large-scale deep learning systems using stochastic computing. In 2016 IEEE International Conference on Rebooting Computing (ICRC) (pp. 1-7). IEEE.

Sim, H., & Lee, J. (2017, June). A new stochastic computing multiplier with application to deep convolutional neural networks. In Proceedings of the 54th Annual Design Automation Conference 2017 (pp. 1-6).

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Venieris, S. I., & Bouganis, C. S. (2016, May). fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (pp. 40-47). IEEE.

Yadan, O., Adams, K., Taigman, Y., & Ranzato, M. A. (2013). Multi-gpu training of convnets. arXiv preprint arXiv:1312.5853.

Yu, S., Zhou, H., Peng, S., Amrouch, H., Henkel, J., & Tan, S. X. D. (2020). Run-Time Accuracy Reconfigurable Stochastic Computing for Dynamic Reliability and Power Management. arXiv preprint arXiv:2004.13320.

Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015, February). Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays (pp. 161-170).

Zhu, J., Wang, L., Liu, H., Tian, S., Deng, Q., & Li, J. (2020). An Efficient Task Assignment Framework to Accelerate DPU-Based Convolutional Neural Network Inference on FPGAs. IEEE Access, 8, 83224-83237.