Fpga Implementation of Precise Convolutional Neural Network for Extreme Learning Machine

Sakthivel R, Suburaaj R

Abstract: ***Feed-forward neural networks can be trained based on a gradient-descent based backpropagation algorithm. But, these algorithms require more computation time. Extreme Learning Machines (ELM's) are time-efficient, and they are less complicated than the conventional gradient-based algorithm. In previous years, an SRAM based convolutional neural network using a receptive – field Approach was proposed. This neural network was used as an encoder for the ELM algorithm and was implemented on FPGA. But, this neural network used an inaccurate 3-stage pipelined parallel adder. Hence, this neural network generates imprecise stimuli to the hidden layer neurons. This paper presents an implementation of precise convolutional neural network for encoding in the ELM algorithm based on the receptive - field approach at the hardware level. In the third stage of the pipelined parallel adder, instead of approximating the output by using one 2-input 15-bit adder, one 4-input 14-bit adder is used. Also, an additional weighted pixel array block is used. This weighted pixel array improves the accuracy of generating 128 weighted pixels. This neural network was simulated using ModelSim-Altera 10.1d and synthesized using Quartus II 13.0 sp1. This neural network is implemented on Cyclone V FPGA and used for pattern recognition applications. Although this design consumes slightly more hardware resources, this design is more accurate compared to previously existing encoders***.

Keywords*: Convolutional Neural Network (CNN), Extreme Learning Machine (ELM), Field Programmable Gate Array (FPGA), Neuromorphic Computing, Pattern Recognition, Receptive-Field (RF), Very-Large Scale Integration (VLSI)

I. INTRODUCTION

The feed-forward neural network is one of the most prevalent among the several types of neural networks. A feed-forward neural network consists of one input layer, one or more hidden layers, and one output layer. It is shown in Fig. 1. Input stimuli from the external sources enter the input layer, and the output signal leaves from the output layer. Feed-forward neural networks are trained based on a

Revised Manuscript Received on June 30, 2020.

***** Correspondence Author
Sakthivel R*, School of Electronics Engg., VIT, Vellore, India. Email: rsakthivel@vit.ac.in
Suburaaj R, School of Electronics Engg., VIT, Vellore, India Email: suburaaj96@gmail.com

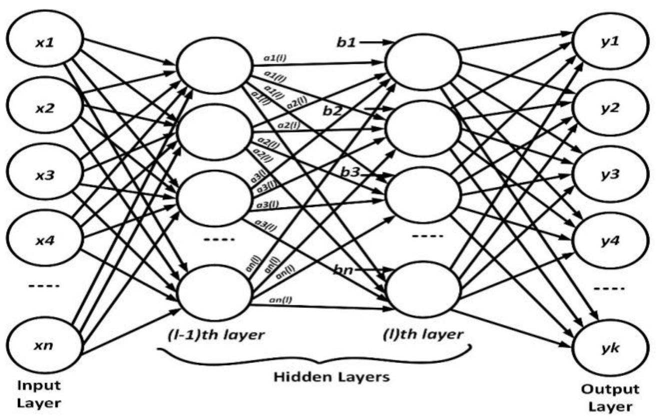


Fig. 1.The architecture of Feed-Forward Neural Network Here, $g(x)$ represents the sigmoid activation function for each node present in the hidden layer.

gradient-descent based backpropagation algorithm [1]. Other training algorithms for training feed-forward neural networks are support vector machines (SVM) [21], perceptrons [22], and radial basis function network learning [23]. These networks most frequently use additive kind of hidden nodes. Hence, the output of the i^{th} node in the l^{th} hidden layer can be expressed as [2]

$$y_i = g\left(a_i^{(l)}.x^{(l)} + b_i^{(l)}\right), \quad b_i^{(l)} \in R \qquad (1)$$

$$g(x) : R \rightarrow R \quad g(x) = 1/(1 + exp\,(-x)) \qquad (2)$$

Here $a_i^{(l)}$ is the weight vector linking the $(l-1)th$ layer to the ith node of the lth layer and bil is the bias of the i^{th} node of the l^{th} layer [2].The architecture of the feed-forward network is shown in Fig. 1. The expression $a_i^{(l)}.x^{(l)}$ signifies the inner vector product of $a_i^{(l)}$ and $x^{(l)}$[2]. But, this algorithm requires more computation time.

Extreme Learning machines (ELM) are time-efficient, and they are less complicated than the conventional gradient-based algorithm. ELM uses weight decay and early stopping methods to prevent problems like local minima and improper learning rate for single-hidden layer feed-forward neural networks [3].ELM can be trained with non-differentiable activation functions contrasting the gradient-based learning algorithms [3]. ELM model is used to predict peptides binding to the Human Leukocyte Antigens (HLA) [24]. ELM model is used in a method predicting quick electricity market value [25]. ELM Models are used in MapReduce ensemble classifier is more exact, efficient, and scalable for analysis of land cover classification data [26]. Section II discusses the literature survey. Sub-section II.

A presents on the ELM algorithm and its training procedure. Sub-section II.B discusses on receptive-field approach in detail. Sub-section II.C discusses the hardware implementation of the RF approach in detail. Section III discusses the technical gap of the previous hardware implementation of an encoder for the Extreme Learning Machine. Section IV discusses the modified methodology of the Convolutional Neural Network for encoding at the hardware level. Section V discusses the pre-processing of data. Sub-section V.A discusses the results obtained after neural network implementation on FPGA. Section VI discusses the comparison of the modified neural network. Section VII briefly discusses the application and future scope of the neural network implemented at the hardware level.

II. LITERATURE SURVEY

A. ELM Algorithm

ELM model is a supervised learning algorithm. ELM is based on a Single-Hidden layer Feed-forward Neural network (SHFN) architecture [3], as shown in Fig. 2. It states the weights connected to the output neurons can be systematically resolved through the generalized inverse operation when the weights and bias of the neurons present in the hidden layer are arbitrarily allocated [4]. They evade the necessity to tune these hidden neuron parameters [4].
Assume that

$$(x_i, t_j) \in \mathbb{R}^n \times \mathbb{R}^m\,; f_N(x_j) = t_j; i = 1, \ldots\ldots, n \text{ and } j = 1, \ldots\ldots, m \qquad (3)$$

where N is the number of patterns, x_i is the input vector, t_i is a output vector. The n and m are the number of the input layer and output layer neurons respectively [4]. If an SLFN with activation function $g(x)$ and N number of hidden layer neurons, there exists B_i, a_i and b_i such that

$$\sum_{i=1}^{N} B_i G(a_i, b_i, x_j) = t_j; j = 1, \ldots\ldots., m \qquad (4)$$

where a_i and b_i are learning parameters of hidden neurons (where a_i is the $n \times 1$ weight vector between the input and the hidden neuron, and b_i is the bias value of hidden neuron), B_i is the i^{th} output weight, and $G(a_i, b_i, x_j)$ is the output of the hidden node for the input vector x_j [4]. If the hidden neuron is additive, it follows that [4]

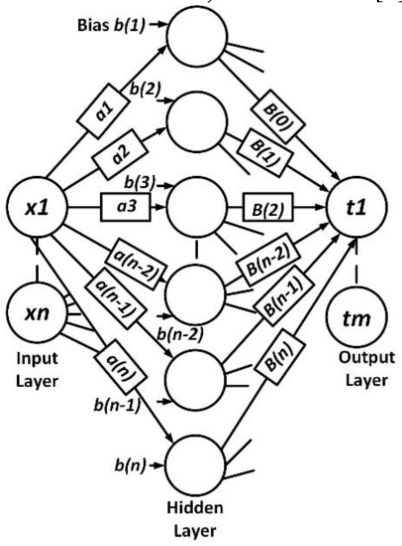


Fig. 2. The architecture of ELM. Adapted from [7].

$$G(a_i, b_i, x_j) = g(a_i.x_j + b_i) \qquad (5)$$

Then, Equation (2) can be written as

$$H.B = T \qquad (6)$$

where

$$H = \begin{bmatrix} G(a_1, b_1, x_1) & \ldots & G(a_N, b_N, x_1) \\ \vdots & \ddots & \vdots \\ G(a_1, b_1, x_N) & \ldots & G(a_N, b_N, x_N) \end{bmatrix} \qquad (7)$$

Input: Training sets $(x_i, t_j) \in \mathbb{R}^n \times \mathbb{R}^m\,; f_N(x_j) = t_j; i = 1, \ldots\ldots, n$ and $j = 1, \ldots\ldots, m$ and N Hidden neurons with sigmoid activation function $g(x)$.

Step 1: Input weight a_i and bias b_i, $i = 1, \ldots, n$ are assigned randomly.

Step 2: The output matrix of the hidden layer (H) is calculated using $G(a_i, b_i, x_j) = g(a_i.x_j + b_i)$ and $H = \begin{bmatrix} G(a_1, b_1, x_1) & \ldots & G(a_N, b_N, x_1) \\ \vdots & \ddots & \vdots \\ G(a_1, b_1, x_N) & \ldots & G(a_N, b_N, x_N) \end{bmatrix}$ [4].

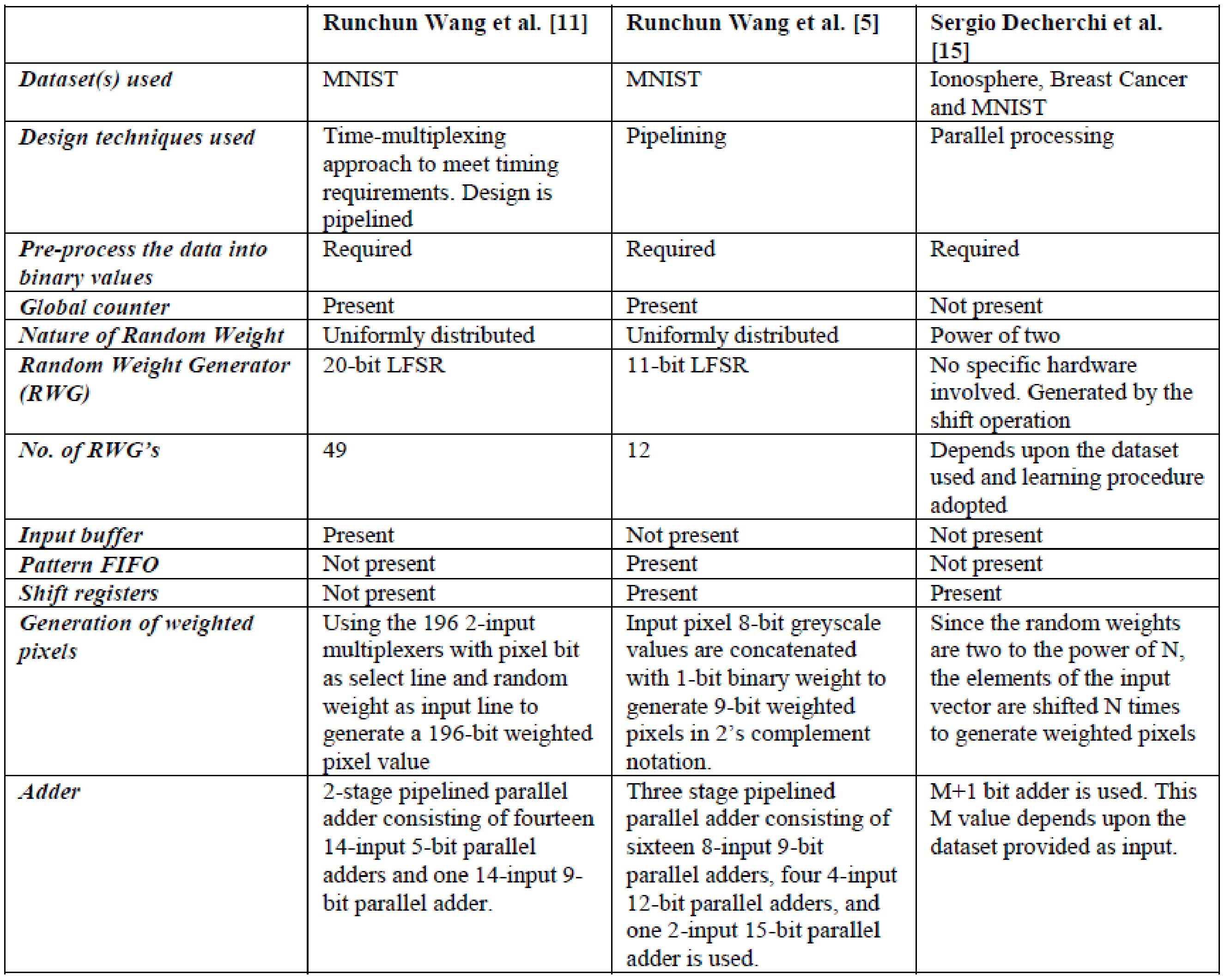
Step 3: Calculate the output weight B using $B = H^+T$ where H^+ is the pseudo-inverse of the hidden layer output matrix H and $T = \begin{bmatrix} t_1 \\ \cdot \\ \cdot \\ t_m \end{bmatrix}$, found using $\sum_{i=1}^{N} B_i G(a_i, b_i, x_j) = t_j; j = 1, \ldots\ldots., m$.

Output: Output weight matrix B

Fig. 3.Training Procedure for Extreme Learning Machine

is the output matrix of the hidden layer [4]. In (7), the i^{th} row is the output of the hidden layer neurons for x_i input vector, and the j^{th} column is the output of the hidden layer neurons for x_1 to x_N input vectors) [4].

Thus, the matrix of output weights, B is estimated as



	Runchun Wang et al. [11]	Runchun Wang et al. [5]	Sergio Decherchi et al. [15]
Dataset(s) used	MNIST	MNIST	Ionosphere, Breast Cancer and MNIST
Design techniques used	Time-multiplexing approach to meet timing requirements. Design is pipelined	Pipelining	Parallel processing
Pre-process the data into binary values	Required	Required	Required
Global counter	Present	Present	Not present
Nature of Random Weight	Uniformly distributed	Uniformly distributed	Power of two
Random Weight Generator (RWG)	20-bit LFSR	11-bit LFSR	No specific hardware involved. Generated by the shift operation
No. of RWG's	49	12	Depends upon the dataset used and learning procedure adopted
Input buffer	Present	Not present	Not present
Pattern FIFO	Not present	Present	Not present
Shift registers	Not present	Present	Present
Generation of weighted pixels	Using the 196 2-input multiplexers with pixel bit as select line and random weight as input line to generate a 196-bit weighted pixel value	Input pixel 8-bit greyscale values are concatenated with 1-bit binary weight to generate 9-bit weighted pixels in 2's complement notation.	Since the random weights are two to the power of N, the elements of the input vector are shifted N times to generate weighted pixels
Adder	2-stage pipelined parallel adder consisting of fourteen 14-input 5-bit parallel adders and one 14-input 9-bit parallel adder.	Three stage pipelined parallel adder consisting of sixteen 8-input 9-bit parallel adders, four 4-input 12-bit parallel adders, and one 2-input 15-bit parallel adder is used.	M+1 bit adder is used. This M value depends upon the dataset provided as input.

Fig. 4.Comparison between blocks present in existing hardware architectures for encoding in Extreme Learning Machine (ELM) algorithm.

$$B = H^{+}T \qquad (8)$$

where H^{+} is the pseudo-inverse of the hidden layer output matrix H [4]. If the matrix H has linearly independent columns, then the expression for calculating H^{+} is

$$H^{+} = (H^{T}H)^{-1}.H^{T} \qquad (9)$$

If the matrix H has linearly independent rows, then the expression for calculating H^{+} is

$$H^{+} = H^{T}.(HH^{T})^{-1} \qquad (10)$$

B. Receptive-Field (RF) Approach

In spite of their efficiency, ELM neural networks are hard to implement at the hardware level because each neuron present in the input layer is linked to all the neurons present in the hidden layer. Hence, the amount of hardware resources increases as the number of hidden layer neurons increases. In [6] and [7], the authors assume a fixed number of neurons present in the input layer and the hidden layer, which limits them from using complex datasets. Hence, an approach based on the receptive field (RF) [8][9] is utilized in implementing a neural network.

The idea of receptive fields [9] usage for pattern recognition originates from neural science, where neurons frequently react to input stimulus generated from a restricted three-dimensional range of data, as shown in Fig. 5. Here,

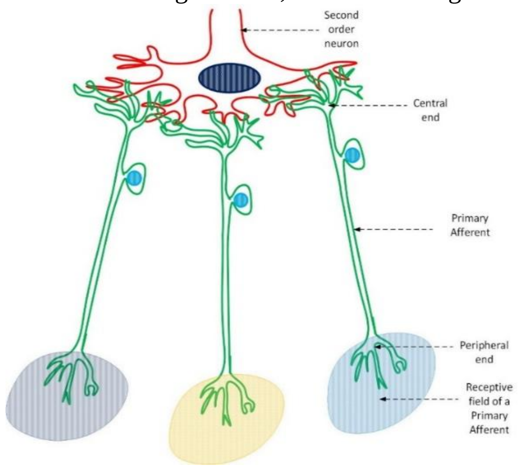


Fig. 5 Receptive field of the second-order neuron [14]

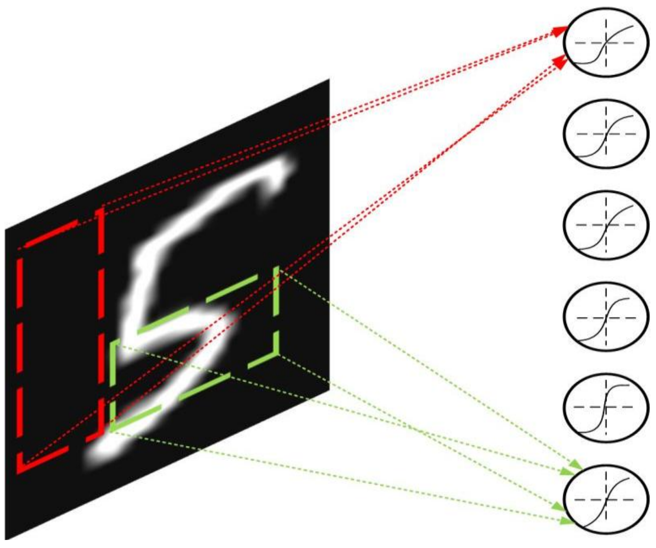


Fig. 6.Illustration of the receptive field methodology [9]. Every single neuron present in the hidden layer obtains inputs from an arbitrary receptive field [9], which is either square or rectangular in size.

Primary Afferent is a pseudo-unipolar neuron with central end synapsing on a second-order neuron, and the peripheral end is connected to receptors. Tissue from which the primary afferent receives sensory information is the receptive field of primary afferent—multiple primary afferents synapsing on second-order neuron (Dorsal Horn Neuron). Hence, the combined receptive field of various primary afferents is the receptive field of the second-order neuron.

The Receptive-field methodology is illustrated, as shown in Fig. 6. Including this concept into hardware implementation for pattern recognition applications improves its performance [9]. Receptive Field-based ELM gives similar accuracy when compared to traditional ELM algorithms [9], with an additional benefit of utilizing fewer hardware resources. But, implementing the RF approach at the hardware level is complex since the location and dimensions of the receptive - field is either found arbitrarily or through patterns, which arbitrarily access the input data.

C. Hardware implementation of RF Approach

Input image in binary format and shift register array is used for hardware implementation of the RF approach (as shown in [5]). Regions are selected from the input image by shifting the index variable across memory array. These generated regions are the receptive-fields. These receptive-fields are either square or rectangular in size.

Fig. 7 shows the functioning of the RF approach on an image from the MNIST dataset [10] of size 28x28 pixels. Other standard datasets used for pattern recognition applications are CIFAR – 10 [16], CIFAR – 100 [16], CALTECH101 [17], CALTECH256 [18] and Image Net [19] respectively. Each pixel can be represented using an 8-bit binary value. First, the image is restructured from 28x28 pixels to 784x1 pixels. Then, RF is shifted by 16 pixels across the restructured image. The 16 pixels corresponds to step size. Here, the size of the receptive field is 128 pixels.
Fig. 7 depicts the restructured image on the left and the region covered by the receptive field over the image on the right for

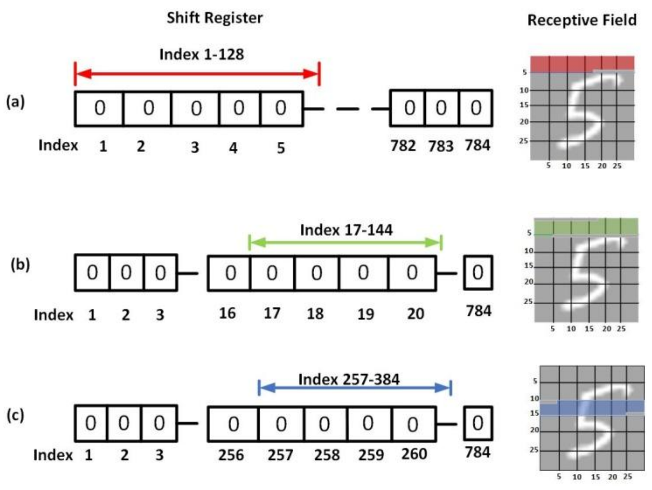

Shift Register
Receptive Field
Index 1-128
(a)
Index 1 2 3 4 5 782 783 784
(b)
Index 17-144
Index 1 2 3 16 17 18 19 20 784
(c)
Index 257-384
Index 1 2 3 256 257 258 259 260 784

Fig. 7.Illustration of RF Approach. Adapted from [5].

different areas covered by the index variable. Fig. 7a shows the receptive-field covered over image for index 1 to index 128 from the restructured image. The region covered by receptive-field (as shown in Fig. 7a) is used for generating stimulus towards the first hidden layer neuron when multiplied by the weights present in the input layer. These weights are arbitrarily generated for each hidden layer neuron. In Fig. 7b, the starting position of the index over a restructured image is shifted by one step, and its corresponding receptive-field is shown. The region covered by receptive-field (as shown in Fig. 7b) is used for generating stimulus towards the second hidden layer neuron when multiplied by the weights present in the input layer.

This process is repeated several times to generate a stimulus to other hidden layer neurons by multiplying with its corresponding binary random weights [5]. Fig. 7c shows the region covered by receptive-field after shifting the starting index by 16 steps over the restructured image. After the 49th step, the final step of the restructured image, the index comes back to its initial position.

III. TECHNICAL GAP

In previous work, the authors presented a huge-parallel system for pattern recognition of handwritten digits, which comprises an input, a hidden, and an output layer, respectively [11]. The input layer is otherwise called an encoder. The hidden layer consists of 8192 neurons. They implemented it on FPGA. In this system, the hardware implementation of the encoder structure is done, assuming all the neurons present in the input layer are connected to all the neurons present in the hidden layer. Under this assumption, more Adaptive Logic Modules are consumed when the system is implemented on FPGA. In [11], the authors explain the disadvantage of storing random weights in Look-Up Table (LUT) as it consumes more memory, and it increases as the number of neurons present in input and hidden layer increases. Sergio Decherchi [15] assumed weights as powers of two and performed multiplication using shift operation. These weights are present between the neurons present in the input and hidden layer. The disadvantage of this approach is these weights are not chosen randomly.

The authors proposed an SRAM based Convolutional Neural Network for encoding using a receptive – field Approach in [5]. This neural network is implemented on FPGA. But it uses an inaccurate parallel adder. Hence, improving accuracy is the motivation of the modified convolutional neural network.

IV. MODIFIED METHODOLOGY

A. Asynchronous FIFO and shift-register array

The topology of Asynchronous FIFO is shown in Fig. 8. It is also used for crossing from the write clock (wclk) domain to the read clock (rclk) domain [20]. It can read and write 8192 pixels. A Shift-register array [5] consisting of eight shift registers is used. These shift registers are connected in a sequential manner. Each shift-register can store 128-bits. The output of asynchronous FIFO [20] is linked to the shift register array at its input. The output of each of these shift registers is connected to the input of a weighted pixel array of size 128 x 9 bits. This weighted pixel array is an additional block that is not present in the previous encoder implemented in [5]. This weighted pixel array improves the accuracy of generating 128 weighted pixels. The input of the parallel adder is connected to the output of 128 9-bit registers from the weighted pixel array.

Assume that asynchronous FIFO is not filled initially. Then, the input pattern is written into asynchronous FIFO stores after its arrival. Then, the initialization of shift registers is done. First, 16 pixels from the asynchronous FIFO are read for each clock cycle, and then the 128-bit shift register is shifted for each clock cycle. This process continues for eight clock cycles.

After eight clock cycles, the register array is shifted by one using the global counter. For example, if there are N hidden layer neurons, then the stimulus is generated for N times. The output to the pipelined parallel adder is generated by a weighted pixel array for every clock cycle with no latency.

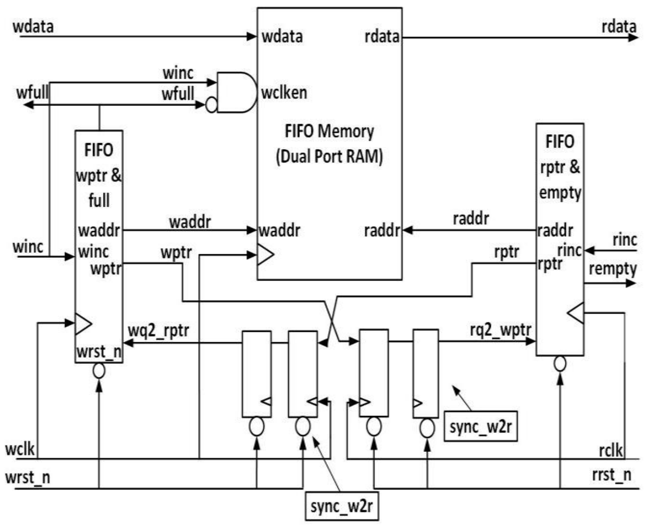


Fig.8 Block Diagram of FIFO [20]

B. Random weight generator and weighted pixel array

The conventional way to calculate weighted pixels is by multiplying the binary pixel value and its corresponding random weights. But the disadvantage of this approach is it

For every pixel value of the input digit image, the random weight generator generates binary weights, which are arbitrary. These binary weights (-1 and 1) are generated based on uniform distribution.

Linear Feedback Shift Registers (LFSR) is used for generating random numbers. LFSR consumes less amount of hardware resources and can be implemented in FPGAs. The LFSR output can be inferred according to 2's complement rule as 0 for +1 and 1 for -1 [5]. An input pixel value is an 8-bit greyscale value. This value is concatenated with the binary weights, resulting in a weighted pixel value [5]. These weighted pixel value is 9-bit value and generated using 2's complement notation requires 128 multipliers, which leads to an increased usage of hardware resources. Hence, the usage of binary weights saves a considerable amount of hardware resources on FPGAs.

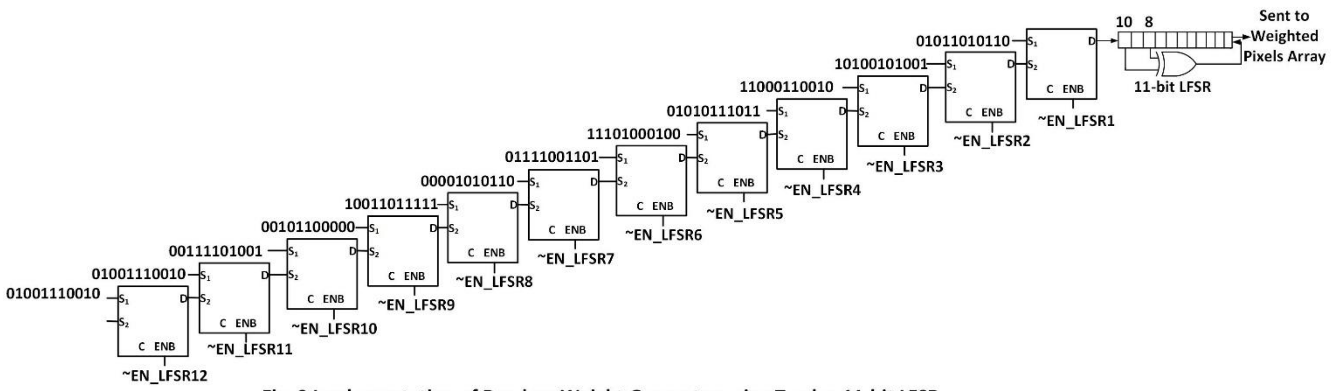


Fig. 9 Implementation of Random Weight Generator using Twelve 11-bit LFSR

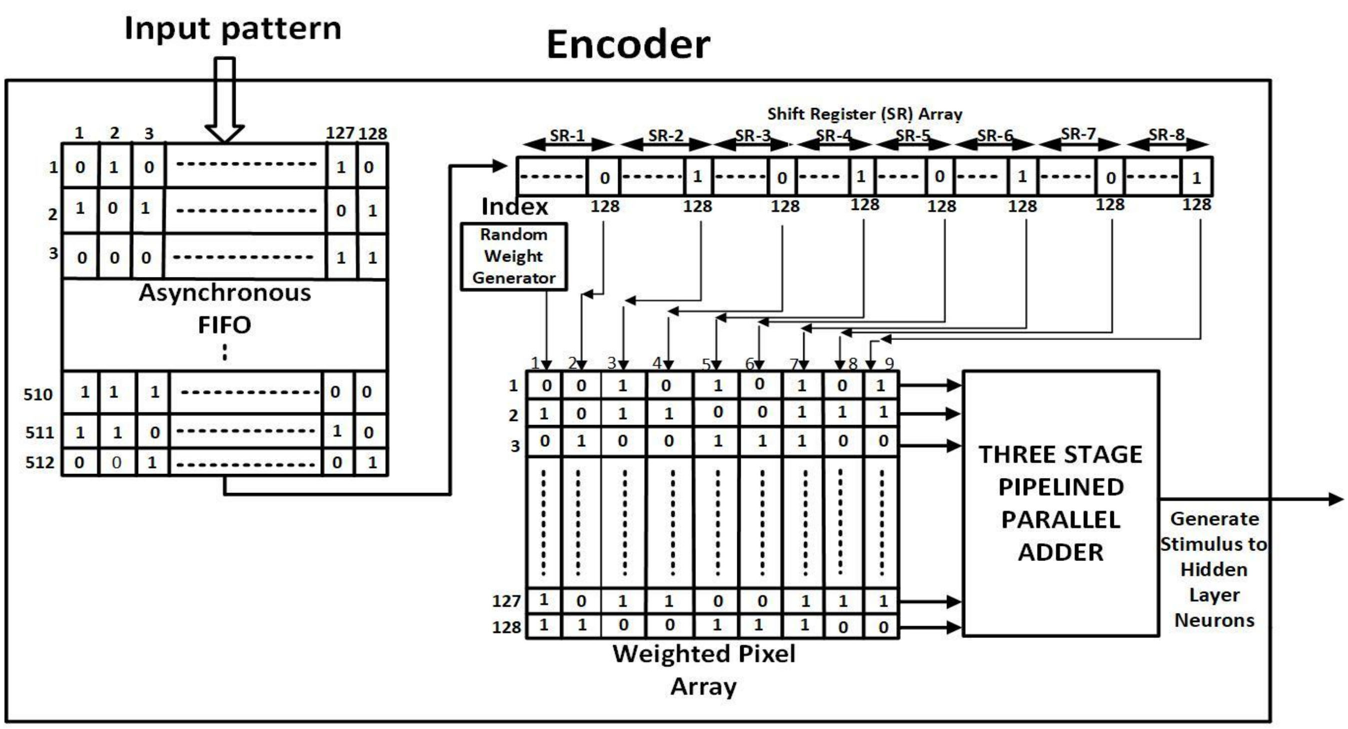


Fig 10. Topology of precise convolutional neural network for encoding

When EN_LFSR1 is high, the corresponding 11-bit binary number is moved into the 11-bit LFSR. Then, it generates random binary weights for the next eleven clock cycles. These weights are concatenated with the pixel values, generating weighted pixels and stored in the weighted pixel array. Then, EN_LFSR2 is high, and the corresponding binary number is moved into the 11-bit LFSR. This process keeps on repeating until EN_LFSR12 becomes high, and then the LFSR generates a random number for the next seven clock cycles. Hence, 128 9-bit weighted pixels are stored in a weighted pixel array.

C. 3 –stage 128-input 9-bit Parallel adder

The stimulus to the neurons present in the hidden layer is generated by adding 128 weighted pixels using 128 input 9-bit Parallel Adder [5]. But, this parallel adder makes a significant delay of nearly 20 ns. Hence, 3-stage pipelined parallel adder is used, as shown in Fig. 10. The first stage involves sixteen 8-input 9-bit adders [5]. The second stage involves four 4-input 12-bit adders [5]. In the third stage, instead of approximating the output of the parallel adder by using one 2-input 15-bit adder [5], one 4-input 14-bit adder is used, as shown in Fig. 11. This modified parallel adder improves the accuracy of the generated stimulus. But, there is a latency of three clock cycles for the generation of stimulus from the output of the encoder. The step by step procedure of the modified methodology is shown in Fig. 12.

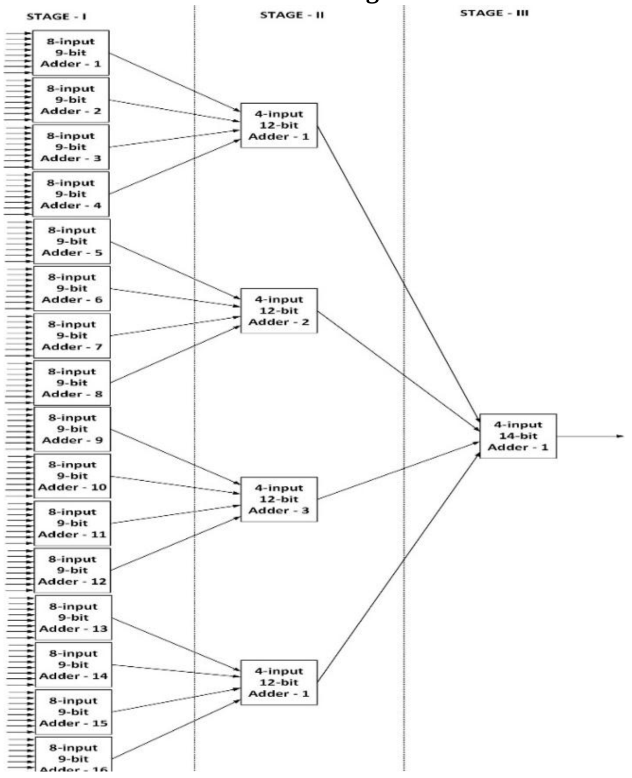


Fig. 11 Structure of three-stage 128- input 9-bit pipelined parallel adder

Input: Images present in the MNIST dataset is processed into 8-bit pixel values and stored in a text file.

Step 1: Assume that asynchronous FIFO is not filled initially. Then, the input data is written into asynchronous FIFO stores after its arrival.

Step 2: Then, the initialization of shift registers is done. First, 16 pixels from the asynchronous FIFO are read for each clock cycle, and then the 128-bit shift register is shifted for each clock cycle. This process continues for eight clock cycles.

Step 3: After eight clock cycles, every 128-bit shift register is shifted by one using the global counter. Since eight 128-bit shift registers are present in an array, 8-bit pixel is generated at the output of shift register array for every clock cycle.

Step 4: The binary random weight is generated from the random weight generator for every clock cycle.

Step 5: Then, the binary random weight is concatenated as MSB with the 8-bit pixel value to generate weighted pixel value. This value is stored in the weighted pixel array.

Step 6: When the 128 weighted pixels are stored in the weighted pixel array, the 3-stage pipelined parallel adder adds these weighted pixels to generate a stimulus.

Output : The generated stimulus

Fig. 12. Step by step procedure of the modified methodology

V. RESULTS

MNIST dataset [10] was converted into 28 × 28 pixel images using Python with Jupyter Notebook. Then, an image file is chosen, and it is converted into 8-bit greyscale values using MATLAB. These 8-bit greyscale values are stored in a text file and provided as design input.

The flow chart of the conversion of the MNIST dataset to the image file is shown in Fig. 13. First, the dataset is downloaded. Since the image and label files are downloaded in archive format, there is a need for extraction. Hence, these files are extracted as u-byte files in .idx format.

These files store different vectors and multi-dimensional matrices. Then, the u-byte files are converted into NumPy n-dimensional arrays. Then, the data present in the NumPy n-dimensional array are saved as images in specific classes and directories. Fig. 14 shows the flowchart of the conversion of an image file to 8-bit greyscale.

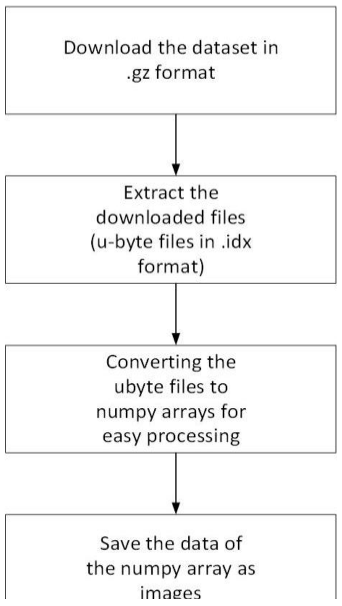

Download the dataset in .gz format
Extract the downloaded files (u-byte files in .idx format)
Converting the ubyte files to numpy arrays for easy processing
Save the data of the numpy array as images

Fig. 13. Flow chart of conversion of MNIST Dataset to an image file

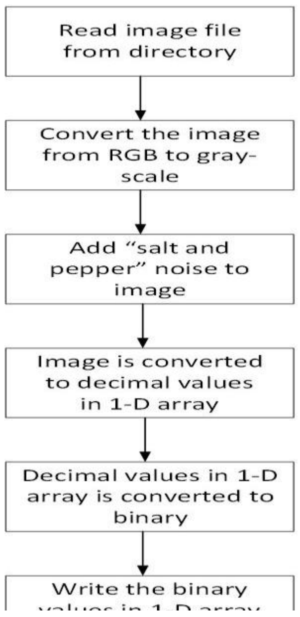

Read image file from directory
Convert the image from RGB to gray-scale
Add "salt and pepper" noise to image
Image is converted to decimal values in 1-D array
Decimal values in 1-D array is converted to binary
Write the binary

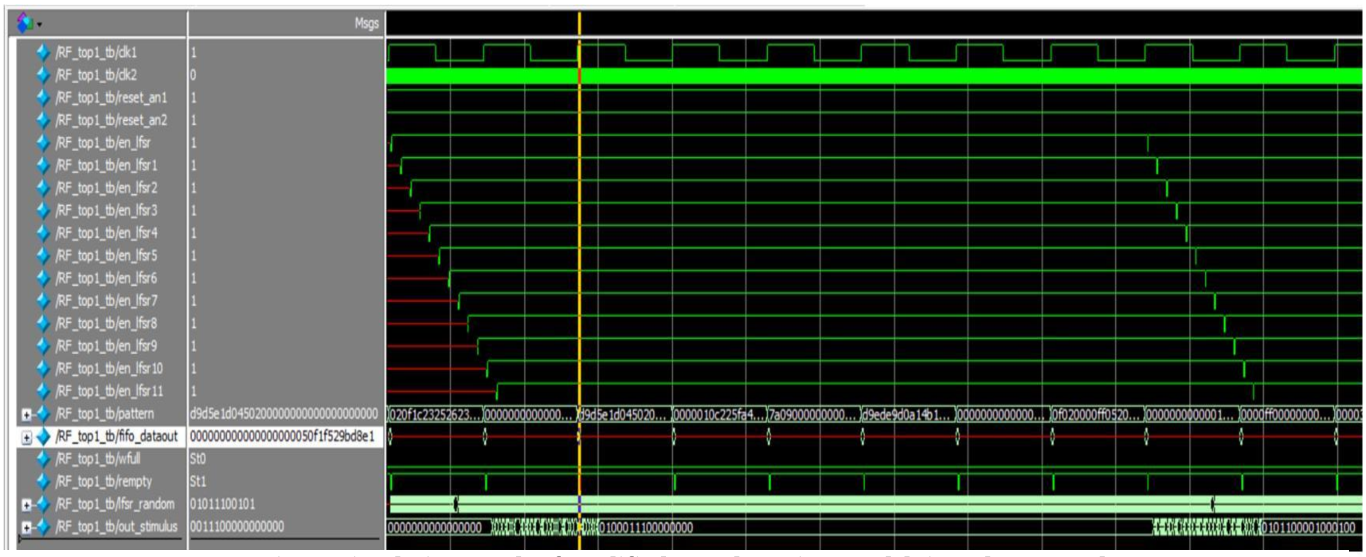
Fig. 14 Flow chart of conversion of an image file to 8-bit greyscale values.

A. FPGA Implementation:

The precise convolutional neural network is written in Verilog. Then, the design is simulated using ModelSim-Altera 10.1d, as shown in Fig. 15. The top module and its sub-modules are described below:

1. RF_top1.v: This module wraps all the sub-modules of the design.

2. RF_generation.v: This sub-module generates the receptive-field from an input image. This receptive-field is generated as an output signal (pattern) of 128-bit wide for every rising edge of the "clk1".



```
                          Msgs
/RF_top1_tb/clk1          1
/RF_top1_tb/clk2          0
/RF_top1_tb/reset_an1     1
/RF_top1_tb/reset_an2     1
/RF_top1_tb/en_lfsr       1
/RF_top1_tb/en_lfsr1      1
/RF_top1_tb/en_lfsr2      1
/RF_top1_tb/en_lfsr3      1
/RF_top1_tb/en_lfsr4      1
/RF_top1_tb/en_lfsr5      1
/RF_top1_tb/en_lfsr6      1
/RF_top1_tb/en_lfsr7      1
/RF_top1_tb/en_lfsr8      1
/RF_top1_tb/en_lfsr9      1
/RF_top1_tb/en_lfsr10     1
/RF_top1_tb/en_lfsr11     1
/RF_top1_tb/pattern       d9d5e1d04502000000000000000000000
/RF_top1_tb/fifo_dataout  00000000000000000050f1f529bd8e1
/RF_top1_tb/wfull         St0
/RF_top1_tb/rempty        St1
/RF_top1_tb/lfsr_random   01011100101
/RF_top1_tb/out_stimulus  0011100000000000
```

Fig.15 Simulation result of modified encoder using ModelSim-Altera 10.1d

3. fifo1.v: This module realizes an asynchronous FIFO logic. This module writes the “pattern” signal into the FIFO for every rising edge of the “clk1”. Then, the 128-bit data is read from an output signal (fifo_dataout) for every rising edge of the clock “clk1”.
4. rwg.v: This sub-module generates an 11-bit random number (lfsr_random) at every rising edge of the “clk2”.
5. weigh_pix_par_add: This sub-module consists of a shift-register array, a global counter, a weighted pixel array, and a parallel adder. This sub-module generates a stimulus as an output signal (out_stimulus).

The input and output port signals are described below:

Input ports:

1. clk1: write clock signal of asynchronous FIFO; clock signal of “RF_generation” and “fifo1” sub-module.
2. clk2: read clock signal of asynchronous FIFO; clock signal of “weigh_pix_par_add” and “rwg” sub-module.
3. reset_an1: Asynchronous negative logic (active-low) reset signal of write clock domain in asynchronous FIFO.
4. reset_an2: Asynchronous negative logic (active-low) reset signal of the read clock domain in asynchronous FIFO.
5. en_lfsr, en_lfsr1…, en_lfsr11 : These are the enable signals of the twelve LFSR’s, which are the inputs to the “rwg” sub-module.

Output ports:

1. pattern: This signal consists of sixteen 8-bit greyscale values. In Fig. 15, this signal is displayed as a 16-bit hexadecimal number. This signal is the output from the “RF_generation” module.
2. fifo_dataout: This signal is the output from the “fifo1” module. In Fig. 15, this signal is displayed as a 16-bit hexadecimal number.
3. wfull: when the contents of asynchronous FIFO is completely written, wfull is 1. This signal is the output signal from the “fifo1” module.
4. rempty: when the contents of asynchronous FIFO is completely read, rempty is 1. This signal is the output signal from the “fifo1” module.

lfsr_random: This signal is an 11-bit binary random number and it is generated from the “rwg” module.

5. out_stimulus: This signal is a 16-bit random number. It is generated from the “weigh_pix_par_add” module.

All the sub-modules of the design are functionally verified. To validate the design, first, the “out_stimulus” signal is generated in decimal format. Then, the “out_stimulus” is calculated by first generating the 128 weighted pixels after zero steps in decimal format and then adding using Microsoft Excel 2013. This calculated value matches with the generated “out_stimulus” signal. Similarly, the 128 weighted pixels with eight steps are generated and then added to calculate “out_stimulus,” using Microsoft Excel 2013.Then, on comparing with the generated “out_stimulus” signal, it makes an error percentage of 0.028%. The generated “out_stimulus” signal and the calculated “out_stimulus” are compared in Fig. 16.

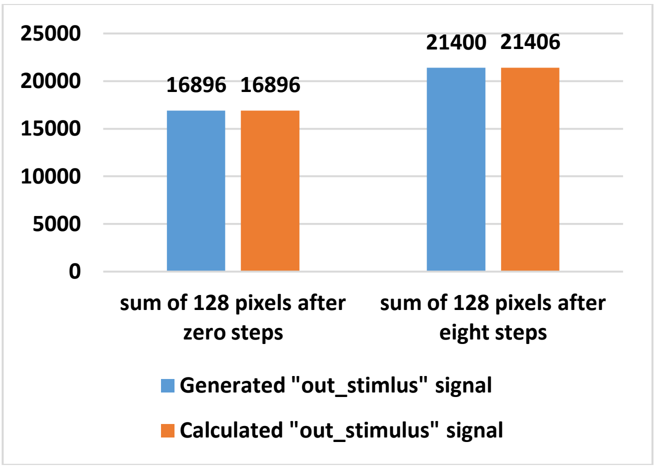


Fig. 16.Comparison of the generated “out_stimulus” and the calculated “out_stimulus” signal.

The frequency of "clk2" is 128 times more than the frequency of "clk1", hence 128 weighted pixels are generated before "fifo_dataout" reads from the asynchronous FIFO. Then, the encoder block was synthesized using Quartus II 13.0 sp1 and implemented on Altera Cyclone V FPGA (Device 5CGXFC5C6F27C7). This FPGA has Adaptive Logic Modules (ALM's) instead of the conventional Logic Elements (LE's) (present in lower level FPGAs) [12]. ALM can add up to 3-bits at a time, while logic elements can add only two bits simultaneously [12] (as shown in Fig. 17). Here, the parallel adder is implemented utilizing ALM's.

The device utilization and maximum clock frequencies of encoder on Cyclone V FPGA are also tabulated in Table I. The logic utilization of encoder is less than 6% of the available adaptive logic modules on Cyclone V FPGA. The total block memory bits used is 1% of the available block memory on Cyclone V FPGA.

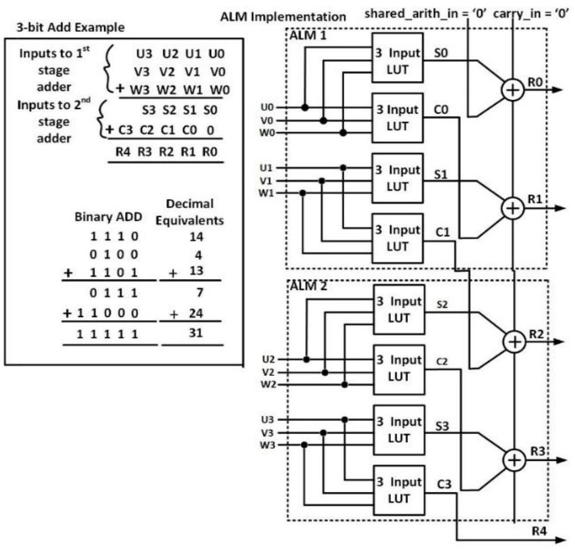


Fig. 17.3-bit addition with Adaptive Logic Module using Shared Arithmetic Mode [13]

Table- I: Device utilization and maximum clock frequencies of the modified encoder

Logic utilization (in ALMs)	1711 / 29,080
Total block memory bits	65536 / 4,567,040
Total DSP Blocks	0 / 150
Total Pins	303 / 364
Total registers	4105
Maximum Clock Frequency (clk1)	234.25 MHz
Maximum Clock Frequency (clk2)	127.76 MHz

VI. COMPARISON OF DESIGN WITH EXISTING ENCODERS

The logic utilization, block memory utilization, the number of DSP Blocks, and the maximum clock frequency of modified encoder are compared with the previously proposed encoders (shown in Fig. 18, Fig. 19, Fig.20 and Fig. 21). The

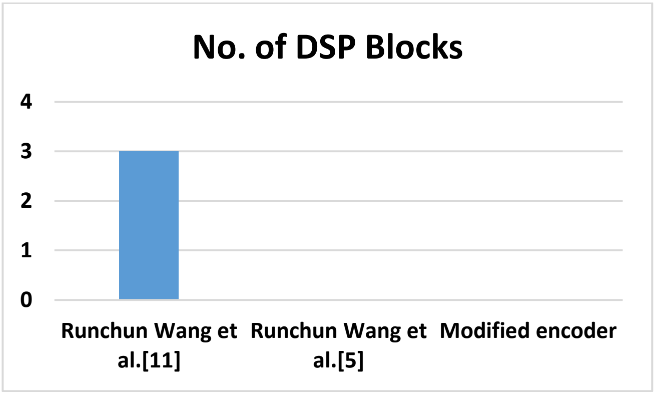


Fig. 18.Comparison on No. of DSP Blocks of the Existing encoders and Modified encoder.

Maximum hardware utilization in FPGA of the modified encoder is compared with the previously proposed encoders (shown in Fig. 22).

The logic utilization of the modified encoder is a little higher compared to Runchun Wang et al. [5]. But, the logic utilization of the modified encoder is less compared to Runchun Wang et al. [11]. The total block memory bits used is the same compared to the Runchun Wang et al. [5]. But, the block memory utilization of the modified encoder is less compared to Runchun Wang et al. [11]. Since no complex multipliers are used, no DSP blocks are used for implementing the modified encoder and Runchun Wang et al. [5]. Since the three 9-bit multipliers are present in the physical neuron, Runchun Wang et al. [11] utilizes three DSP blocks. The maximum clock frequency is a little lower compared to Runchun Wang et al. [5] and Runchun Wang et al. [11], but higher than Sergio Decherchi et al. [15]. The Maximum hardware utilization in FPGA of the modified encoder is lower than Runchun Wang et al. [11], slightly higher than Runchun Wang et al. [5] and higher than Sergio Decherchi et al. [15].

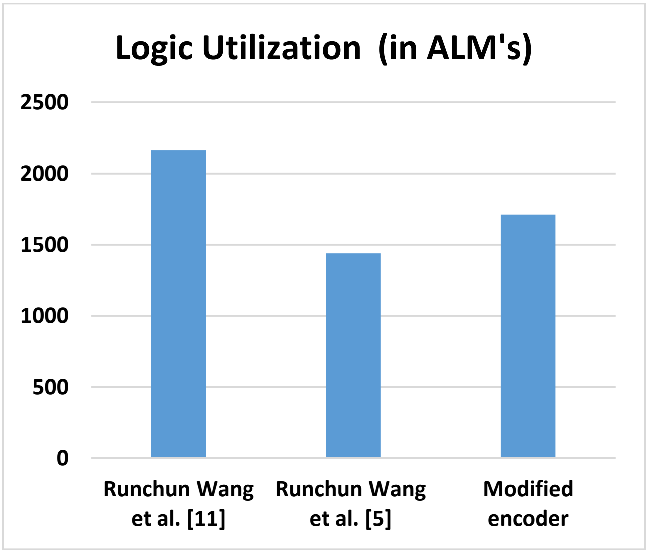


Fig. 19.Comparison of Logic Utilization of the Existing encoders and Modified encoder.

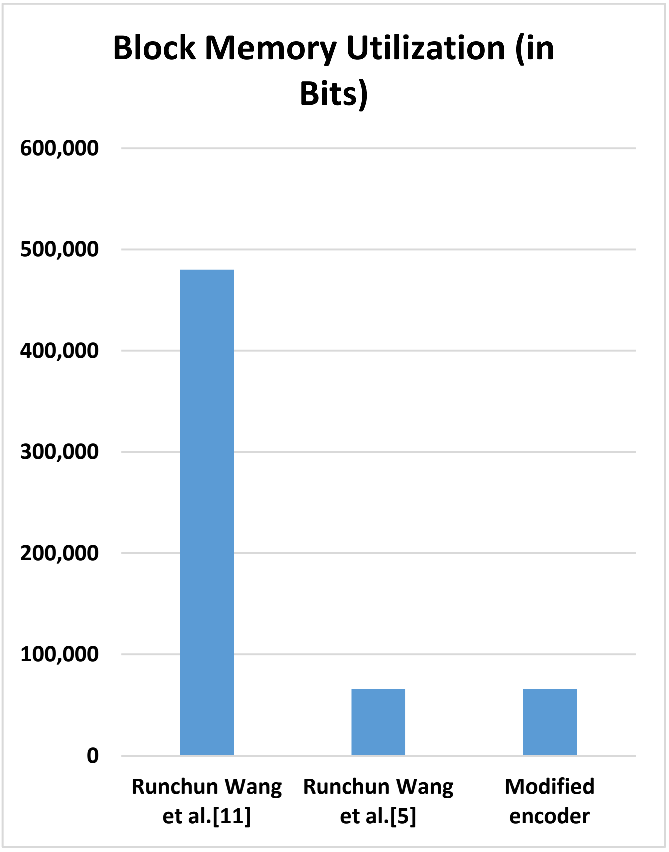


Fig. 20.Comparison of Block Memory Utilization of the Existing encoders and Modified encoder.

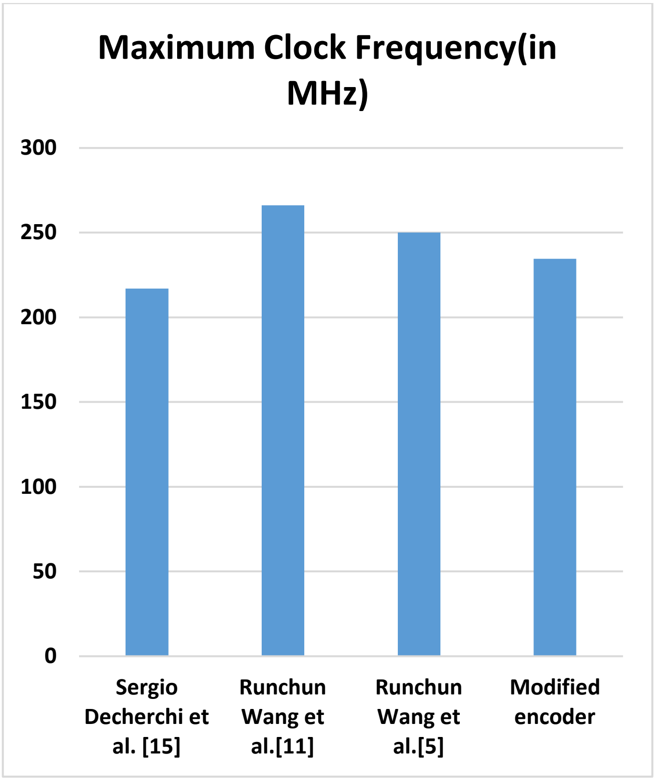


Fig. 21.Comparison of the Maximum Clock Frequency between the Existing encoders and Modified encoder.

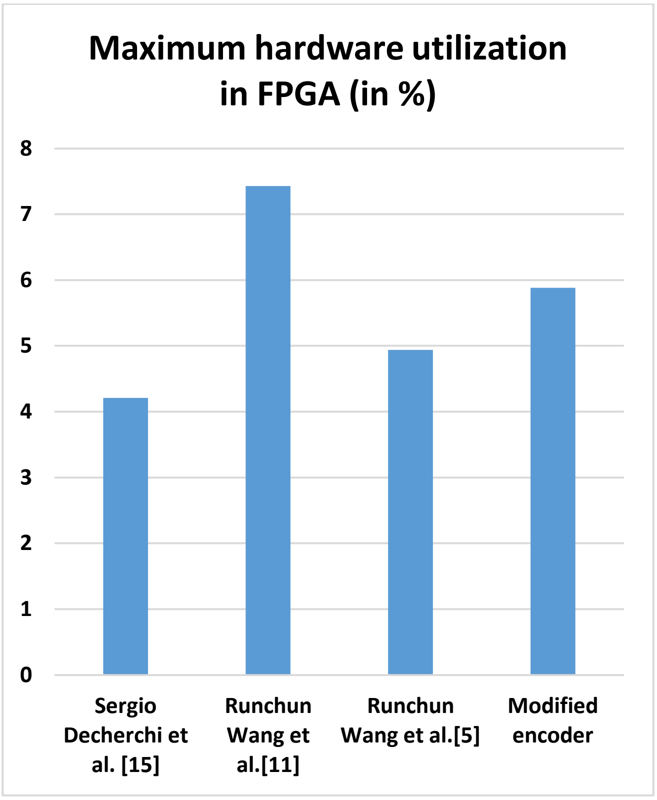


Fig. 22.Comparison of the Maximum hardware utilization between the Existing encoders and Modified encoder

VII. FUTURE SCOPE

This encoder can be used for building large – scale neural network system. Since reconfigurable hardware logics are used, the system can be implemented on FPGA. This system consists of an input, a hidden, and an output layer, respectively. The modified convolutional neural network acts as an input layer for encoding. The hidden layer consists of 128 neural cores, and each core consists of N number of neurons. These neurons are implemented at the hardware level using multipliers and adders. The neurons present in the output layer can be implemented using the accumulator and register, which adds the output of hidden layer neurons multiplied by a weight. This weight is present between hidden layer neuron and output layer neuron.

VIII. CONCLUSION

We have implemented a Precise Convolutional Neural Network for encoding in Extreme Learning Machine Algorithm based on Receptive – Field Approach (RF) at the hardware level. The weighted pixel array block improves the accuracy of generating 128 weighted pixels, but it increases the amount of hardware utilized in FPGA. The 3-stage pipelined parallel adder is modified to improve the accuracy of the stimulus generated from the output of the encoder. Although this design consumes slightly more hardware resources, this design is more accurate compared to previously existing encoders. Now, we can implement a large-scale neural network system for pattern recognition applications.

REFERENCES

1. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by backpropagation errors. Nature 323:533–536.
2. G.-B. Huang, D. H. Wang, and Y. Lan, “Extreme learning machines: a survey,” *International Journal of Machine Learning and Cybernetics*, vol. 2, no. 2, pp. 107–122, May 2011.
3. G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Extreme learning machine: Theory and applications,” *Neurocomputing*, vol. 70, no. 1–3, pp. 489–501, Dec. 2006.
4. J. V. Frances-Villora, A. Rosado-Munoz, M. Bataller-Mompean, J. Barrios-Aviles, and J. F. Guerrero-Martinez, “Moving Learning Machine towards Fast Real-Time Applications: A High-Speed FPGA-Based Implementation of the OS-ELM Training Algorithm,” Electronics, vol. 7, no. 11, p. 308, Nov. 2018.
5. R. Wang, G. Cohen, S. Thakur, J. Tapson, and A. Van Schaik, “An SRAM-based implementation of a convolutional neural network”, 2017.
6. C. S. Thakur, R. Wang, T. J. Hamilton, J. Tapson, and A. van Schaik, “A Low Power Trainable Neuromorphic Integrated Circuit That Is Tolerant to Device Mismatch,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 2, pp. 211–221, Feb. 2016.
7. Thakur, Chetan Singh, T. J. Hamilton, R. Wang, J. Tapson, and van Schaik, “A neuromorphic hardware framework based on population coding,” *2015 International Joint Conference on Neural Networks (IJCNN)*,2015.[Online].Available:https://www.academia.edu/17633387/A_neuromorphic_hardware_framework_based_on_population_coding.
8. G.-B. Huang, Z. Bai, L. L. C. Kasun, and C. M. Vong, “Local Receptive Fields Based Extreme Learning Machine,” *IEEE Computational Intelligence Magazine*, vol. 10, no. 2, pp. 18–29, May 2015.
9. M. D. McDonnell, M. D. Tissera, T. Vladusich, A. van Schaik, and J. Tapson, “Fast, Simple and Accurate Handwritten Digit Classification by Training Shallow Neural Network Classifiers with the ‘Extreme Learning Machine’ Algorithm,” *PLOS ONE*, vol. 10, no. 8, p. e0134254, Aug. 2015.
10. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
11. R. Wang, C. S. Thakur, G. Cohen, T. J. Hamilton, J. Tapson, and A. van Schaik, “Neuromorphic Hardware Architecture Using the Neural Engineering Framework for Pattern Recognition,” *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 3, pp. 574–584, Jun. 2017.
12. “Quartus II Handbook Version 13.1”, Altera Corporation, 2013
13. “Stratix III Device Handbook”, Altera Corporation, 2013.
14. J. Hunter, “receptive field,” *www.youtube.com*. [Online]. Available: https://www.youtube.com/watch?v=6xs8FF8A1F0. [Accessed: 26-Feb-2020].
15. S. Decherchi, P. Gastaldo, A. Leoncini, and R. Zunino, “Efficient Digital Implementation of Extreme Learning Machines for Classification,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 8, pp. 496–500, Aug. 2012.
16. A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” 2009.
17. L. Fei-Fei, R. Fergus and P. Perona, “Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories”, *IEEE. CVPR 2004*, Workshop on Generative-Model Based Vision. 2004.
18. Griffin, G. Holub, AD. Perona, P., “The Caltech 256**”,** Caltech Technical Report.
19. “ImageNet”, *image-net.org*. [Online]. Available: http://image-net.org/download-images. [Accessed: 07-Mar-2020].
20. C. E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” *SNUG San Jose*, 2002.
21. C. Cortes and V. Vapnik, “Support-vector networks,” Machine Learning, vol. 20, no. 3, pp. 273–297, Sep. 1995.
22. F. Rosenblatt, Principles of neurodynamics: perceptrons and the theory of brain mechanisms. New York: Spartan Books, 1962.
23. D. Lowe, “Adaptive radial basis function nonlinearities and the problem of generalisation,” in *Proceedings of first IEE international conference on artificial neural networks*, 1989, pp. 171–175.
24. S. Handoko, K. Chee Keong, O. Yew, G. Zhang, and V. Brusic, “Extreme Learning Machine for Predicting HLA-Peptide Binding,” LNCS, vol. 3973, pp. 716–721, 2006.
25. X. Chen, Z. Y. Dong, K. Meng, Y. Xu, K. P. Wong, and H. W. Ngan, “Electricity Price Forecasting With Extreme Learning Machine and Bootstrapping,” IEEE Transactions on Power Systems, vol. 27, no. 4, pp. 2055–2062, Nov. 2012
26. J. Chen, G. Zheng, and H. Chen, “ELM-MapReduce: MapReduce Accelerated Extreme Learning Machine for Big Spatial Data Analysis,” in IEEE International Conference on Control and Automation (ICCA), 2013, pp. 400–405.

AUTHORS PROFILE

Dr.R.Sakthivel received Bachelor degree in Electrical Engineering from Madras University in 2000 and his M.E degree in Applied Electronics from Anna University in 2004. He has received his Doctorate in the area of Low Power High speed architecture development for signal processing and cryptography. He is currently working as an Associate Professor in the School of Electronics Engineering at Vellore Institute of Technology University, Vellore. His research area includes Low power, high speed architectures, Hardwares for AI and ML. He is the Co-author of Basic Electrical Engineering" Published by Sonaversity in the year 2001 and author of VLSI Design published by S.Chand in the year 2007. He has also published several technical papers in national and international conferences/ Journals. He has delivered around 50 Guest lectures / Invited talk and hands on workshop in the area of FPGA based System Design, Analog IC Design, Fulll Custom IC Design, RTL to GDSII, ASIC Design etc.

Suburaaj R is pursuing his Master degree in Very Large Scale Integrated (VLSI) Design under the Dept. of Micro- and Nano-electronics, School of Electronics Engineering at Vellore Institute of Technology, Vellore. He has received his Bachelor degree in Electronics and Instrumentation from Amrita School of Engineering at Amrita Vishwa Vidyapeetham, Coimbatore in 2017. He has published one IEEE conference paper. His research area is Area-efficient architectures for machine learning algorithms at hardware level.