Dynamically Growing Neural Network Architecture for Lifelong Deep Learning on the Edge

Duvindu Piyasena*, Miyuru Thathsara†, Sathursan Kanagarajah‡, Siew-Kei Lam§ and Meiqing Wu¶
*,†,§, ¶Nanyang Technological University, Singapore
{*gpiyasena, §assklam,¶meiqingwu}@ntu.edu.sg, †mthathsara@outlook.com, ‡ksathursan1408@gmail.com

***Abstract*—Conventional deep learning models are trained once and deployed. However, models deployed in agents operating in dynamic environments need to constantly acquire new knowledge, while preventing catastrophic forgetting of previous knowledge. This ability is commonly referred to as lifelong learning. In this paper, we address the performance and resource challenges for realizing lifelong learning on edge devices. We propose a FPGA based architecture for a Self-Organization Neural Network (SONN), that in combination with a Convolutional Neural Network (CNN) can perform class-incremental lifelong learning for object classification. The proposed SONN architecture is capable of performing unsupervised learning on input features from the CNN by dynamically growing neurons and connections. In order to meet the tight constraints of edge computing, we introduce efficient scheduling methods to maximize resource reuse and parallelism, as well as approximate computing strategies. Experiments based on the Core50 dataset for continuous object recognition from video sequences demonstrated that the proposed FPGA architecture significantly outperforms CPU and GPU based implementations.**

***Keywords*-Deep learning, lifelong learning, continual learning, incremental learning, self-organization, FPGA**

I. Introduction

Recent advancements in deep learning and computing technology have enabled deep learning models to be deployed in a wide range of embedded applications for self-driving cars, autonomous robots, etc [1], [2]. Such applications typically encounter settings that are not available during training. Ideally, the models should be able to acquire new knowledge in real-time, while not interfering with previous knowledge. This ability is referred to as *lifelong learning*[1] which is now an active area of research [3]. The following are desired for lifelong learning in embedded applications.

Real-time learning. In conventional deep neural networks (DNNs), learning is performed on large data over many hours or days. However, a lifelong learning model needs to learn quickly from a few samples in real-time.

Learning at the edge. Deep learning models are usually trained using GPUs in the cloud, which has several limitations, e.g., 1) network connectivity cannot always be guaranteed, 2) privacy concerns of sending data to cloud. Performing learning on the edge device overcomes these problems.

Custom computing. Embedded systems have tight computational and memory constraints, and they often run on strict power budgets. Lifelong learning algorithms deployed on the edge will need design strategies that take full advantage of the compute capabilities of the target platform.

Meeting the above requirements is challenging. Firstly, it has been shown that when connectionist neural networks are trained incrementally with new knowledge, the models undergo *catastrophic forgetting* of previous knowledge [4]. This can be solved naively by storing all the past data and shuffling them with new data when training on new tasks. However, it is not practical to store such large datasets due to memory resource constraints and privacy concerns. Secondly, back-propagation based learning that is used in state-of-the-art deep learning models require cloud grade GPU or specialized accelerators like TPU [5], due to high compute and memory requirements. Despite this, training a model from scratch still takes a long time ranging from hours to a few days. In addition, these cloud grade computing platforms are ill-suited for deployment in edge devices due to their high energy consumption. These challenges can only be overcome with new deep learning models capable of unsupervised lifelong learning and custom hardware implementations of the same.

In this paper, we propose an FPGA based custom hardware architecture for class-incremental lifelong learning, targeting object classification task in edge devices. The proposed model is based on a Self-Organization Neural Network (SONN), called *Grow When Required (GwR)* artificial neural network [6]. GwR network leverages on local learning and self-organization with dynamic neuron/connection formation/pruning. We combined a backbone CNN network as a feature extractor with the GwR based SONN as a classifier to achieve lifelong learning, and propose a novel FPGA architecture for the SONN. Our main contributions are:

- Novel hardware design of SONN that performs class-incremental learning for object classification. This is the first work to propose a FPGA architecture for dynamically growing SONN, where the neurons and connections evolve over time. Previously reported hardware implementations of SONN have fixed neurons and connections.
- Efficient scheduling methods to maximize resource reuse and parallelism. Bit-width quantization and approximate computing are used to reduce hardware complexity.
- Proposed FPGA architecture significantly outperforms CPU and GPU implementations on the Core50 dataset for continuous object recognition from video streams.

[1]The terms *continual learning, incremental learning* are also used commonly in literature.

II. Related Work

A. Lifelong Learning

Lifelong learning refers to learning new tasks efficiently, while not catastrophically forgetting previous tasks. The current approaches fall under three categories: *1) Regularization 2) Architectural 3) Memory Rehearsal* [3], [7]. Regularization approaches penalize changes to parameters important to previous knowledge by modifying the regular loss [8], [9], [10]. Architectural approaches dynamically change the model architecture over time to accommodate new knowledge [11], [12]. In memory rehearsal methods, subset of training data are buffered and replayed when learning new tasks [13], [14].

B. Self-Organization and Lifelong Learning

Our work adopts an architectural approach to address catastrophic forgetting with a SONN. SONN is a family of artificial neural networks, which is used for topology learning of a feature space through an unsupervised learning method named competitive learning [6], [15], [16]. Various types of SONNs have been proposed in literature, with the most popular one being the Self-Organized Map (SOM) [15], However, the fixed network size and time-decaying learning rule limit its ability to learn continuously. The work in [16] proposed to periodically grow a network of neurons. GwR [6] extends [16] by replacing the periodic neuron adding criteria with a rule that adds neurons based on network activity. Recently, SONNs have been used in combination with CNNs to perform lifelong learning for object classification [12], [17]–[19]. Particularly inspired by [12], we combine a CNN with a SONN based on GwR to demonstrate the feasibility of performing lifelong learning under edge constraints using custom hardware acceleration.

C. FPGA based lifelong learning

The works in [20], [21] propose to accelerate conventional DNN training using multi-FPGA platforms. Recent works [22]–[24] propose FPGA based lightweight models particularly for lifelong learning. The work in [22] relies on a Hebbian learning rule, while [23], [24] rely on a product quantization strategy for lifelong learning. Similar to the proposed work, these methods leverage on local learning and learns quickly from few samples. However, contrary to our work, they rely on a fixed set of neurons, which limits the ability to autonomously learn from non-stationary data distributions.

III. Proposed Lifelong Learning Model

The proposed model, shown in Fig. 1, combines a Convolutional Neural Network (CNN) and a SONN, where the latent features extracted from the CNN are fed into the SONN that acts as the classifier. As the model is exposed to new classes, the SONN evolves while CNN layers remain static.

CNN models trained on large datasets (e.g., Imagenet [25]) have been shown to perform well, as generic feature extractors [26]. We have used the Resnet-18 CNN model, pre-trained on Imagenet. The latent features are extracted from the last convolutional layer (*Avg Pool*) of Resnet-18 of dimension (D_f) 512, and the subsequent Fully-Connected layer is discarded. The SONN is based on the GwR network [6]. In this paper, we will only discuss the hardware architecture of the classifier (i.e. SONN), since there are already many works on hardware implementation of CNNs [27]–[30].

A. SONN Algorithm

The SONN performs unsupervised learning on input features and creates a discrete topology representation of the feature space using a dynamically growing set of neurons (N) and connections (C). During training, neurons are added progressively to better represent input topology, while connections are formed between similar neurons following the Hebbian learning principle [31].

Each neuron ($i \in N$) has several parameters: *a) weights vector (w_i)*, which determines neuron position in the feature space, b) habituation counter (H_i), measure of how frequent the neuron has fired, c) *class probability vector (p_i)*, probability of association to all the classes, and d) *connectivity graph* ($G(V_i, E_i)$), graph of connected neurons (V_i) and connections (E_i). Each connection ($c \in C$) has an associated age (age_c).

Algorithm 1 Proposed SONN pseudo-code

```
1:  [Step 1 : Winner Selection Phase (WSEL)]
2:  for $i$ = 1 to $N_{curr}$ do // inter-neuron loop
3:      for $j$ = 1 to $D_f$ do // intra-neuron loop
4:          $dist_i \leftarrow dist_i + |x_j - w_{i,j}|$
5:      end for
6:      if $dist_i < dist_{b_1}$ then
7:          $\{b_1, dist_{b_1}\}, \{b_2, dist_{b_2}\} \leftarrow \{i, dist_i\}, \{b_1, dist_{b_1}\}$
8:      else if $dist_i < dist_{b_2}$ then
9:          $\{b_1, dist_{b_1}\}, \{b_2, dist_{b_2}\} \leftarrow \{b_1, dist_{b_1}\}, \{i, dist_i\}$
10:     end if
11: end for
12: if ($dist_{b_1} > dist_T$ and $H[h_{b_1}] < h_T$) then
13:     [Step 2 : Neuron Addition Phase (NADD)]
14:     for $j = 1 \ldots D_f$ do // intra-neuron loop
15:         $w_n \leftarrow (x_j + w_{b_1,j})/2$
16:     end for
17:     $h_n \leftarrow 0$ , $p_{n,l} \leftarrow 1$, and $p_{n,j} \leftarrow 0, j \neq l$
18:     Connect $n$ with $b_1$, $b_2$ and disconnect $b_1$, $b_2$
19: else
20:     [Step 3 : Neuron Train Phase (NTRAIN)]
21:     for $i = 1 \ldots len(neigh_{b_1})$ do // inter-neuron loop
22:         for $j = 1 \ldots D_f$ do // intra-neuron loop
23:             $w_{i,j} \leftarrow w_{i,j} + \epsilon \cdot H[h_i] \cdot (x_j - w_{i,j})$
24:         end for
25:         $h_i \leftarrow h_i + 1$ , $p_{n,l} \leftarrow p_{n,l} + 1$
26:     end for
27: end if
```

Algorithm 1 decribes the training of SONN, which consists of three main steps: a) Winner Selection (*WSEL*), b) Neuron Add (*NADD*), and Neuron Train (*NTRAIN*).

1) Step 1. Winner Selection (WSEL): Upon receiving an input feature vector (x), SONN computes the Manhattan distance between input (x) and weights vectors (w) of all neurons and the 2 neurons with the least distance to the input

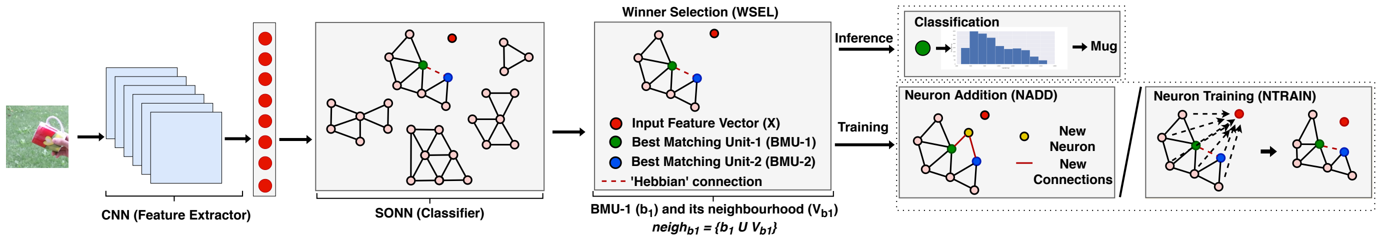


Fig. 1: Proposed Model

are selected as best matching units, BMU-1 (b_1) and BMU-2 (b_2). The class with maximum probability of association ($argmax(p_{b_1})$) is predicted as the classification output. During training, if BMU-1 is both distant from input ($dist > dist_T$) and sufficiently mature ($H[h_{b_1}] < h_T$), a new neuron is added to SONN to better represent the input (Step 2). Consequently, neurons that are sufficiently dissimilar to the input and well habituated are not modified, preventing catastrophic interference to consolidated knowledge of the network. On the other hand, if this condition is not met, BMU-1 and its connected neighbour neurons ($neigh_{b_1}$ = $b_1 \bigcup \mathrm{V}_{b_1}$) are trained.

2) Step 2. Neuron Add (NADD): If a neuron is added, it (n) is assigned weights (w_n) associated with BMU-1 (w_{b_1}) and input feature vector (x). The class probability vector of the neuron (p_n) is initialized to associate with the class of the training sample (l). The new neuron is connected to both BMU-1 and BMU-2, and if a connection existed between BMU-1 and BMU-2, it is removed. The changes to the *connectivity graphs (G(V, E))* of the three neurons are shown in Eq. 1.

$$\begin{aligned} &V_{b_1} = V_{b_1} \bigcup \{n\}/\{b_2\}, V_{b_2} = V_{b_2} \bigcup \{n\}/\{b_1\}, \\ &E_{b_1} = E_{b_1} \bigcup \{(n, b_1)\}/\{(b_1, b_2)\}, \\ &E_{b_2} = E_{b_2} \bigcup \{(n, b_2)\}/\{(b_1, b_2)\}, \\ &V_n = \{b_1, b_2\}, E_n = \{(n, b_1), (n, b_2)\} \end{aligned} \tag{1}$$

3) Step 3. Neuron Train (NTRAIN): If a neuron is not added, then the weights of BMU-1 and its connected neighbours ($\{w_i | i \in neigh_{b1}\}$) are modified such that they are closer to the input in the feature space, and their habituation counters are updated to decrease response of neurons to further training. Next, the age of all connections of BMU-1 (E_{b_1}) are increased by a step, except the one between BMU-1 and BMU-2 (b_1, b_2). If the latter is present, $age_{(b_1,b_2)}$ is reset to zero. Otherwise, a new connection is created with age set to zero. Finally, the class probability vector of BMU-1 (p_{b_1}) is updated to increase its association to class label of training input.

As the SONN is adapted only at the local neighbourhood that is most similar to the input ($neigh_{b_1}$), catastrophic interference to parameters unrelated to the current input is prevented. This local adaptation is lightweight in terms of computation and memory requirements, compared to back-propagation based learning with entire model adaptation. The dynamically growing nature of the SONN enables the model to adapt to non-stationary data streams and grow accordingly. Most importantly, the proposed SONN has inherent parallelism of vector computations which lends itself well towards custom hardware implementations.

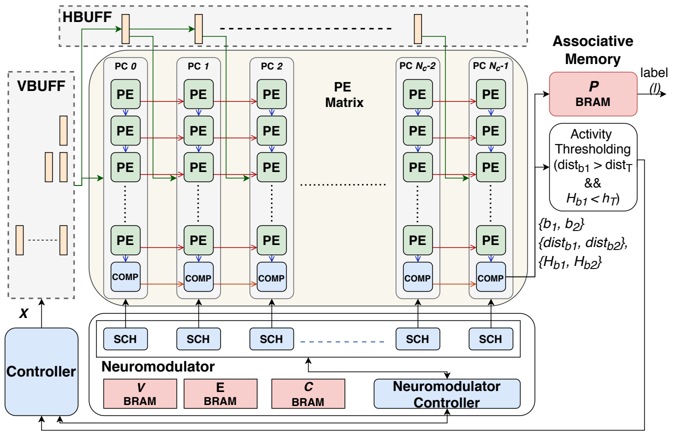


Fig. 2: Overview of proposed architecture

IV. Proposed Hardware Architecture

A. Overview of Architecture

Fig. 2 shows the proposed architecture, which consists of four main blocks: *PE Matrix*, *Neuromodulator*, *Associative Memory* and *Controller*. *PE Matrix* performs vector computations of neuron weights: distance calculation of input to weights of all neurons and winner selection during *WSEL*, calculation of new neuron weights during *NADD*, and updating weights during *NTRAIN*. *Neuromodulator* schedules the operations on the PE matrix during training (*NTRAIN* and *NADD*), while maintaining neighbourhood graphs of all neurons and state of connections. *Associative memory* maintains the probability scores of each class per each neuron. These three blocks are controlled and synchronized by the *Controller*.

B. Hardware Approximations

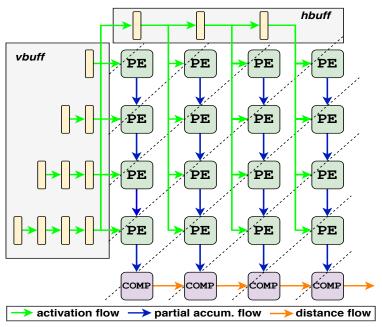
The following approximations were applied to various operations of the GwR algorithm.

Quantization. Inputs and parameters (originally in floating-point) were quantized to 8-bit integers [32]. In *NTRAIN*, the learning rate was rounded to nearest power-of-2 values to replace costly multiplications with bit-shift operations.

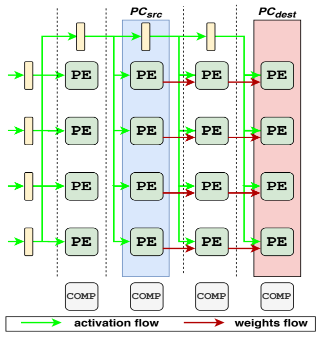
Distance Computation. The Euclidean distance (d_i=$||x - w_i||$) in GwR is replaced with the relatively lighter Manhattan distance, d_i=$||x - w_i||_1$.

Activity Calculation. In GwR, the activity of BMU-1 (act = $\exp(-d_{b_1})$) is thresholded ($act < act_T$) to decide on the addition of a new neuron. To avoid the exponential calculation, this was replaced with distance thresholding ($dist_{b_1} > dist_T$).

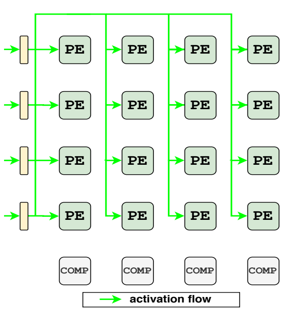
Habituation Counter. Each time a neuron is trained during *NTRAIN*, its habituation counter is updated by $\tau \cdot 1.05 \cdot$ (1-$H[h_i]$) - τ, (τ is a constant). To reduce the number of expensive arithmetic operations at runtime, we precomputed



(a) *WSEL* step (Pipelined array)



(b) *NADD* step (Semi-broadcast array)



(c) *NTRAIN* step (Full-broadcast array)

Fig. 3: PE Matrix dataflow

100 discrete values offline. These values were quantized to 8-bit and stored in a ROM (H). A pointer (h) is maintained per neuron to point to its current value in H, which is incremented each time the neuron is trained (Algorithm 1, line 25).

C. Exploiting Parallelism

Parallelism is exploited for the vector processing operations involving neuron weights (w) in *WSEL*, *NADD* and *NTRAIN*. These operations have 2 kinds of parallelism: *inter-neuron* and *intra-neuron* parallelism, which refer to concurrent operations between neurons and within a neuron respectively. Both parallelisms are achieved in the proposed hardware architecture by unrolling *inter-neuron* loops (Algorithm 1, lines 2, 21) and *intra-neuron* loops (lines 3, 14, 22) by N_c and N_r factors respectively. The unrolled design is executed in a systolic array of processing elements (PE Matrix). Each PE is connected with adjacent PEs supporting pipelined dataflow as shown in Fig. 3. This allows pipeline execution of both inter-neuron and intra-neuron loops, which results in further parallelism.

D. Functional Units

1) ***PE Matrix****:* PE matrix consists of a set of Processing Columns (PC), which operates under three different configurations: a) Pipelined array (*WSEL*), b) Semi-broadcast array (*NADD*), and c) full broadcast array (*NTRAIN*). Each configuration supports different dataflow requirements in the three steps while maximizing parallelism, as shown in Fig. 3. Each SONN neuron is organized as a Processing Column (PC). A PC contains an array of processing elements (PE), multipliers to perform computations, and a memory bank for storing weights (w) and habituation pointers (h) of its local neurons, as shown in Fig. 4a. When neurons are progressively added to SONN, they are assigned a PC in a modulo round-robin manner. Thus, when the m^{th} neuron is added, $m \mod N_c^{th}$ PC will store its weights (w_m) and habituation pointers (h_m), and all its computations as per Algorithm 1 will be performed locally. Due to round robin assignment, the neurons are evenly distributed across the PCs, with each PC supporting up to $\lceil N_{max}/N_c \rceil$ neurons. The inter-neuron loops are unrolled and executed in parallel by the PCs, while intra-neuron loops are unrolled and executed in parallel within each PC.

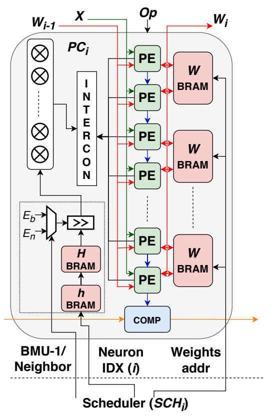
During *WSEL*, each PC computes distance between input (x) and neuron weights (w) that are local to the column. Each PE computes distance between an input and a weight vector element followed by an accumulation of distance from its upstream neighbor. This operation of the j^{th} PE of i^{th} PC is described in line 4 of Algorithm 1.

As shown in Fig. 3a, in each PC, the partial accumulated distance values are sent downstream (blue arrows). To synchronize with upstream partial accumulated inputs, *vbuff* delays input to each PE row in the PC. The final accumulated distance is sent to a local comparator unit (*Comp*), where it gets compared with the two current BMU-1 and BMU-2 distances ($dist_{b_1}, dist_{b_2}$) received from downstream comparators. BMU-1 and BMU-2 are updated according to Algorithm 1, lines 6-10. The updated BMU-1 and BMU-2 are forwarded to the next comparator (orange arrows). To synchronize with comparator dataflow, input to each PC is delayed by *hbuff*.

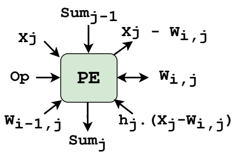
During *NADD*, the PC containing BMU-1 ($PC_{src} = b_1 \mod N_c$) computes the weights of new neuron added (n) as described by Algorithm 1, lines 14-16. The computed weights (w_n) are transferred from the PC_{src} to PC_{dest}, by shifting them horizontally across the PE matrix from one column to another. This is shown in Fig. 3b, where red arrows represent the movement of weights. The weights are eventually stored in PC_{dest}, which is determined by the neuromodulator.

In *NTRAIN*, neurons in the set $neigh_{b_1}$ are modified according to Algorithm 1, Step 3, by the corresponding PCs belonging to the set PC_{train}=$\{x \mod N_c | \forall x \in neigh_{b_1}\}$ in parallel, while other PCs are idle. If a PC have multiple neurons to be trained, updates are queued and performed sequentially by that PC. Each PE in a PC performs weights update of a single weight vector element independently of other PEs. The behaviour of j^{th} PE when PC updates the i^{th} neuron weights is shown in Algorithm 1, line 23. The PE computes difference between x and w vector elements ($x_j - w_{i,j}$), and a multiplier bank local to each PC performs $\epsilon \cdot H[h_i] \cdot (x_j - w_{i,j})$, and the result is added to $w_{i,j}$ inside PE.

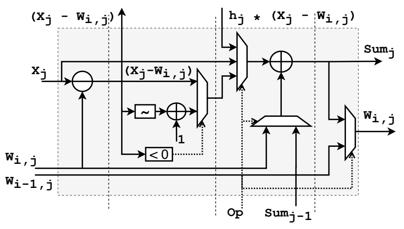
Fig. 4c shows the internal structure of a PE, where the different computations required for the three steps are performed using shared arithmetic resources. The operation of the PE is controlled by the controller and neuromodulator, depending on the operational steps and the role of the corresponding PC. Based on the resource constraints, intra-neuron loops of Algorithm 1 are folded by a factor of D_f/N_r. Weights and habituation pointers of neurons are stored in BRAM blocks dedicated to each PC, so that the computations are not delayed by external memory accesses. Each PC requires $8 \cdot D_f \cdot \lceil N_{max}/N_c \rceil$ and $8 \cdot (100 + \lceil N_{max}/N_c \rceil)$ bits for weights



(a) PC structure



(b) PE block



(c) PE internal structure

Fig. 4: Processing Column (PC) structure

and habituation storage respectively.

*2) **Neuromodulator**:* During *NADD* and *NTRAIN*, only a subset of PCs are active at a time. The neuromodulator schedules computations to these required PC(s) during these two steps. To control individual PCs, the neuromodulator consists of a scheduler block with N_c scheduler units (*sch*), each connected to a PC as shown in Fig. 2. The neuromodulator also maintains *connectivity graphs (G)* for all SONN neurons in the form of lists of neighbour neurons (V) and connections (E) of each neuron, in two separate BRAM based memories.

In *NADD*, PC_{src} computes the new neuron weights (w_n) and shift them through PCs to PC_{dest}. The individual scheduler units set PC_{src} to compute new neuron weights, PC_{dest} to store the weights, PCs in between these to bypass the weights, and other PCs to idle. Additionally, the connectivity graphs of BMU-1 (V_{b_1}) and BMU-2 (V_{b_2}) are updated in the V and E memories, while a new entry ($G(V_n, E_n)$) is created in each of them for the new neuron, as explained in Eq. 1.

In *NTRAIN*, neuromodulator schedules PCs in the set PC_{train} to train $neigh_{b_1}$ set of neurons. For this, neighbour neuron list of BMU-1, V_{b_1}, is read from V memory and the memory bits containing $neigh_{b_1}(b_1 \bigcup V_{b_1})$ set of neurons is scanned by all N_c scheduler units in parallel. After which, each scheduler unit identifies neuron(s) associated with the connected PC. The set of neurons to be trained in the j^{th} PC is denoted as N_j=$\{n|n \mod N_c$=$j, \forall n$∈$neigh_{b_1}\}$. Consequently, PC_j updates N_j neurons iteratively as per Algorithm 1, line 23, while each PC operates independently in parallel.

The design assumes maximum neighbours of any neuron to be $Neigh_{max}$, which is empirically determined. Each neuron and connection is identified by an identifier of $log_2(N_{max})$ and $log_2(C_{max})$ bits respectively. Therefore the on-chip BRAM storage requirement for the two memories, V and E is $8{\cdot}N_{max}{\cdot}Neigh_{max}{\cdot}(log_2(N_{max})+log_2(C_{max}))$.

*3) **Associative Memory**:* This unit (P) maintains the class probability vectors (p) of all neurons on BRAM, which is updated during NADD and NTRAIN phases as per Algorithm 1, lines 17, 25. A single probability is represented by 8 bits. Hence, memory requirement of P, is $8{\cdot}N_{classes}{\cdot}N_{max}$ bits.

*4) **Controller**:* The Controller manages the dataflow of the PE matrix in the three steps by feeding activations into the PE Matrix in a synchronous manner, and synchronizing PE Matrix and neuromodulator operations.

E. Latency and Storage Analysis

As shown in Fig. 5, as neurons are added, the latency of *WSEL* increases linearly in a piece-wise manner due to the N_c inter-neuron parallelism of PE Matrix. The latencies of *NADD* and *NTRAIN* do not increase monotonically with neurons. *NADD* latency is a function of PC_{src} and PC_{dest}, due to the pipelined data movement explained earlier. The latency of *NTRAIN* is dependant on the maximum neurons to be trained in a single column (max_{sch}). The mean and standard deviation of the latency distributions of *NADD* and *NTRAIN* are summarised in Table II.

The proposed design assumes an upper-bound of neuron growth and connection formation, with maximum limit of neurons N_{max}, neighbours per neuron $Neigh_{max}$, and connections C_{max}. The bounds are constrained by the availability of on-chip BRAM, and could be extended by storing neuron parameters in an off-chip memory. The total storage requirement of the design is given by Eq. 2.

$$S_T = 8{\cdot}[\underbrace{N_{max}{\cdot}(D_f+1)+100{\cdot}N_c}_{\text{PE Matrix}} + \underbrace{N_{classes}{\cdot}N_{max}}_{\text{assoc. mem}}] + \underbrace{[N_{max}{\cdot}Neigh_{max}{\cdot}(log_2(N_{max})+log_2(C_{max})) + 8{\cdot}C_{max}]}_{\text{Neuromodulator}} \tag{2}$$

V. Experimental Results

We analyze and evaluate our proposed method on Core50, a popular benchmark dataset developed specifically for continuous object recognition from video sequences [33]. The dataset consists of 50 object classes belonging to 10 categories, with classification performed at both levels. We use the train/test split suggested in [33], but images are sampled at 1fps from the video stream as consecutive images are similar. The model was evaluated under New Classes (NC) scenario, where model was trained sequentially in a class-incremental manner.

The following baselines are used. A vanilla Resnet18 model was trained with stochastic gradient descent (SGD) in a class-incremental manner (*Naive*) and using entire dataset at once (*Cumulative*). Original GwR model (*GwR*) was used to evaluate the proposed approximations. Additionally, the proposed model is compared with two popular lifelong learning methods for class incremental learning (*iCaRL* [13] and *ExStream* [34]).

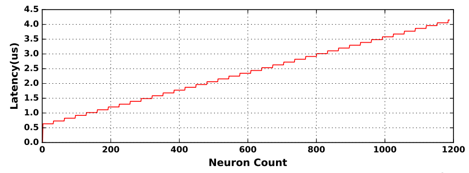
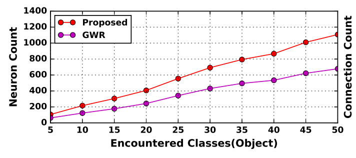
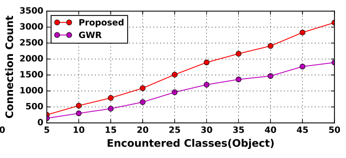


Fig. 5: *WSEL* latency, ($Freq.$ = $200Mhz$)

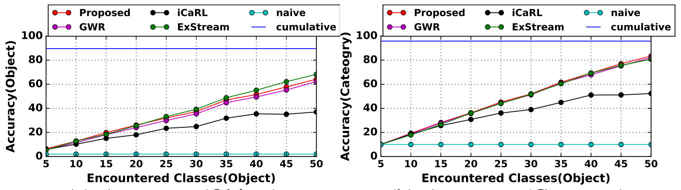


(a) Neuron Growth



(b) Connection Growth

Fig. 6: Model growth comparison



(a) Accuracy (Object) (b) Accuracy (Category)

Fig. 7: Model behaviour

TABLE I: Model performance evaluation

Method	Accuracy (%)		Train Time	Mem. requirement
	Object	Category	(LL/total) s	(Params + Data, *MB*)
Naive[e]	2	10	-	-
Cumulative[f]	89.64	95.86	-	-
ExStream[b] [34]	**67.45**	81.68	70.92/139.75	1.906 + 1.563
iCaRL[a] [13]	36.99	52.33	2359.4/2359.4	136 + 75
GwR[c]	62.472	82.3	37.37/85.46	1.53 + 0
Proposed[d]	64.25	**83.45**	**20.17/66.69**	**0.73 + 0**

[a] *replay buffer size* = 400, *lr*=0.005
[b] *replay buffer size* = 800, *network/train params adopted from* [34]
[c] ϵ_{b_1} = 0.5, $\epsilon_{v_{b_1}}$ = 0.07, τ_b=0.3, τ_n=0.1 act_T=0.55, h_T=0.1
[d] ϵ_{b_1} = 0.5, $\epsilon_{v_{b_1}}$ = 0.0625, τ=0.3, $dist_T$=205, h_T=0.1
[e,f] *batch size*=16, *epochs:naive*=1, *cumulative*=10, *learn. rate*=0.001

A. Accuracy evaluation

The approaches are evaluated based on accuracy, training time and memory requirement. The results are summarised in Table I. The train times were measured using models developed on PyTorch framework [35] and executed on GPU, while memory requirement was calculated from trainable parameters, and data storage, required for lifelong learning. It can be observed that *Naive* undergoes *catastrophic forgetting*. Among the lifelong learning models, the proposed SONN exhibits higher accuracy than iCaRL and ExStream in category level, while executing faster and consuming lesser memory. Although, the proposed SONN generates more neurons and connections than *GwR* as shown in Fig. 6, the memory requirement and runtime is lesser, with slight improvement in accuracy. Hence, the proposed model is clearly more amenable to edge implementation with tight computation and memory constraints. Fig. 7 shows the accuracy change of all methods as classes are encountered.

B. Hardware Comparison

The proposed SONN was implemented with Verilog HDL and synthesized using Xilinx Vivado 2019.1 for Zynq UltraScale+ ZCU9EG at 200Mhz. CPU and GPU implementations of the SONN was developed using PyTorch framework [35], for benchmarking the proposed hardware.

Table II shows the performance of all devices for the three steps separately per video sample. The range of *WSEL* latency with neuron growth is used for comparison among devices, which shows that the FPGA design outperforms CPU and GPU by 229 and 166 times respectively. The *NTRAIN* time for each sample depends on the size of the set $neigh_{b_1}$ for CPU/GPU implementations, and the distribution of $neigh_{b_1}$ across PCs for FPGA. To be fair, we compare the mean distribution of latencies for *NTRAIN* samples. Results show that FPGA outperform CPU and GPU by 779 and 1000 respectively. The comparison of *NADD* latencies show FPGA with gains of 2389 and 3539 times over CPU and GPU respectively. The FPGA design significantly outperforms the CPU and GPU implementations due to 1) application specific parallelism and pipelining described in Section IV, and 2) distributed on-chip BRAM memory offering high bandwidth. During *NADD* and *NTRAIN*, CPU outperforms the GPU. A significantly higher degree of parallelism is required for the GPU to gain a runtime advantage over the faster CPU clock operating frequency and the synchronization overhead with the host CPU.

TABLE II: Latency (*us*) comparison for CPU, GPU and FPGA

Step	GPU[a]	CPU[b]	FPGA[c]
WSEL (min-max)	570 - 691	204 - 950.8	0.63 - 4.145
NTRAIN (mean ± std)	1220 ± 224	734 ± 224	0.484 ± 0.157
NADD (mean ± std)	1600 ± 720	1080 ± 350	0.487 ± 0.144

[a] Nvidia GTX 1080, 2560 CUDA cores, 1.7GHz, 8GB
[b] Intel Xeon E5-1650v2, 3.50GHz, 12MB cache, 16GB RAM, 6 OMP threads
[c] 200Mhz, N_{max}=2048, C_{max}=8096, $Neigh_{max}$=29, N_r=27, N_c=32

TABLE III: Hardware results and comparison

	[23]	[24]	[22]	Proposed
Design Parameters				
Feature Extractor	Inception-V3	Inception V3	-	Resnet-18
Feat. Vec. dim	2048	2048	100	512
No. of Neurons	200	30	1000	2048
Output Classes	10	10	10	50
Network Params	409610	61450	1000000	1150976
Bit-Precision	16-bit	16-bit	16-bit	8-bit/ 16-bit
Mem. Requirement	1.30E+07	1.11E+07	1.60E+07	1.22E+07
Synthesis Results				
Device	xc7vx690tffg	xcvu13P	xc7vx690T	xzcu9eg
LUT	265680	152546	232,111	115656
FF	-	-	∼5000	104680
DSP	2048	2064	3000	1184
BRAM (36kb)	-	-	-	415.5
Max Frequency (Mhz)	209	204	93	200

Table III compares the hardware resource consumption of the proposed design with existing FPGA based lifelong learning models with local learning rules. Similar to the proposed work, the existing implementations exclude feature extractors. The number of neurons in the hidden layer is presented along with memory requirement for each work. It can be observed that the proposed work consumes the least resources. A major drawback common to all these existing methods is that they assume that the number of neurons and neurons per class are known a-priori. As such, unlike all the existing hardware based lifelong learning models, our proposed SONN is capable of growing dynamically to adapt to non-stationary data streams.

VI. Conclusion

Lifelong learning on the edge is essential for autonomous agents operating in dynamic environments that need constant model adaptation. We proposed a FPGA based SONN that dynamically evolves to adapt to non-stationary data streams. In order to maintain scalability and reduce runtime, an efficient scheduling method is employed to maximize resource reuse and parallelism. We show that the proposed hardware approximation strategies not only resulted in shorter training time and lesser memory storage, but also improves accuracy. The proposed FPGA architecture also significantly outperforms the CPU and GPU implementations due to the effective use of application-specific parallelism and on-chip BRAM.

VII. Acknowledgement

This work was supported in part by the National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) Programme with the Technical University of Munich at TUMCREATE.

References

[1] L. Fridman, D. E. Brown, M. Glazer, W. Angell, S. Dodd, B. Jenik, J. Terwilliger, A. Patsekin, J. Kindelsberger, L. Ding, S. Seaman, A. Mehler, A. Sipperley, A. Pettinato, B. D. Seppelt, L. Angell, B. Mehler, and B. Reimer, "Mit advanced vehicle technology study: Large-scale naturalistic driving study of driver behavior and interaction with automation," *IEEE Access*, vol. 7, pp. 102 021–102 038, 2019.

[2] G. Angeletti, B. Caputo, and T. Tommasi, "Adaptive deep learning through visual domain localization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 7135–7142.

[3] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54 – 71, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608019300231

[4] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," ser. Psychology of Learning and Motivation, G. H. Bower, Ed. Academic Press, 1989, vol. 24, pp. 109 – 165. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0079742108605368

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and et al., "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[6] S. Marsland, J. Shapiro, and U. Nehmzow, "A self-organising network that grows when required," *Neural Networks*, vol. 15, no. 8, pp. 1041 – 1058, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608002000783

[7] T. Lesort, V. Lomonaco, A. Stoian, D. Maltoni, D. Filliat, and N. D. Rodríguez, "Continual learning for robotics," *ArXiv*, vol. abs/1907.00182, 2019.

[8] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 12, pp. 2935–2947, Dec 2018.

[9] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, 2017. [Online]. Available: https://www.pnas.org/content/114/13/3521

[10] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *ICML*, 2017.

[11] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," *arXiv preprint arXiv:1708.01547*, 2017.

[12] G. I. Parisi, J. Tani, C. Weber, and S. Wermter, "Lifelong learning of spatiotemporal representations with dual-memory recurrent self-organization," *Frontiers in neurorobotics*, vol. 12, p. 78, 2018.

[13] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, "icarl: Incremental classifier and representation learning," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 2001–2010.

[14] D. Lopez-Paz and M. Ranzato, "Gradient episodic memory for continual learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 6467–6476.

[15] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[16] B. Fritzke, "A growing neural gas network learns topologies," in *Advances in neural information processing systems*, 1995, pp. 625–632.

[17] A. Gepperth and C. Karaoguz, "A bio-inspired incremental learning architecture for applied perceptual problems," *Cognitive Computation*, vol. 8, no. 5, pp. 924–934, 2016.

[18] J. L. Part and O. Lemon, "Incremental online learning of objects for robots operating in real environments," in *2017 Joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, Sep. 2017, pp. 304–310.

[19] ——, "Towards a robot architecture for situated lifelong object learning," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Nov 2019, pp. 1854–1860.

[20] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2016, pp. 107–114.

[21] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt, "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 81–84.

[22] A. R. Daram, D. Kudithipudi, and A. Yanguas-Gil, "Task-based neuromodulation architecture for lifelong learning," in *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2019, pp. 191–197.

[23] G. B. Hacene, V. Gripon, N. Farrugia, M. Arzel, and M. Jezequel, "Budget restricted incremental learning with pre-trained convolutional neural networks and binary associative memories," *Journal of Signal Processing Systems*, vol. 91, no. 9, pp. 1063–1073, 2019.

[24] ——, "Efficient hardware implementation of incremental learning and inference on chip," 2019.

[25] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[26] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: An astounding baseline for recognition," in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, June 2014, pp. 512–519.

[27] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[28] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[29] D. Piyasena, R. Wickramasinghe, D. Paul, S. Lam, and M. Wu, "Reducing dynamic power in streaming cnn hardware accelerators by exploiting computational redundancies," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 354–359.

[30] ——, "Lowering dynamic power of a stream-based cnn hardware accelerator," in *2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP)*, 2019, pp. 1–6.

[31] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[32] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[33] V. Lomonaco and D. Maltoni, "Core50: a new dataset and benchmark for continuous object recognition," *arXiv preprint arXiv:1705.03550*, 2017.

[34] T. L. Hayes, N. D. Cahill, and C. Kanan, "Memory efficient experience replay for streaming learning," in *2019 International Conference on Robotics and Automation (ICRA)*, May 2019, pp. 9769–9776.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf