Pre-Defined Sparse Neural Networks With Hardware Acceleration

Sourya Dey, Kuan-Wen Huang, Peter A. Beerel, *Senior Member, IEEE*, and Keith M. Chugg, *Fellow, IEEE*

***Abstract*— Neural networks have proven to be extremely powerful tools for modern artificial intelligence applications, but computational and storage complexity remain limiting factors. This paper presents two compatible contributions towards reducing the time, energy, computational, and storage complexities associated with multilayer perceptrons. Pre-defined sparsity is proposed to reduce the complexity during both training and inference, regardless of the implementation platform. Our results show that storage and computational complexity can be reduced by factors greater than 5X without significant performance loss. The second contribution is an architecture for hardware acceleration that is compatible with pre-defined sparsity. This architecture supports both training and inference modes and is flexible in the sense that it is not tied to a specific number of neurons. For example, this flexibility implies that various sized neural networks can be supported on various sized field programmable gate array (FPGA)s.**

***Index Terms*— Machine learning, neural network, multilayer perceptron, sparsity, hardware acceleration.**

I. Introduction

NEURAL networks are critical drivers of new technologies such as computer vision, speech recognition, and autonomous systems. As more data have become available, the size and complexity of neural network (NN)s has risen sharply, with modern NNs containing millions or even billions of trainable parameters [1], [2]. These massive NNs come with the cost of large computational and storage demands. The current state of the art is to train large NNs on Graphical Processing Unit (GPU)s in the cloud – a process that can take days to weeks even on powerful GPUs [1]–[3] or similar programmable processors with multiply-accumulate accelerators [4]. Once trained, the model can be used for inference, which is less computationally intensive and is typically performed on more general purpose processors (*i.e.,* Central Processing Unit (CPU)s). It is increasingly desirable to run inference, and even some re-training, on embedded processors which have limited resources for computation and storage. In this regard, model reduction has been identified as a key to NN acceleration by several prominent researchers [5]. This is generally performed post-training to reduce the memory requirements to store the model for inference – *e.g.,* methods for quantization, compression, and grouping parameters [6]–[9].

Decreasing the time, computation, storage, and energy costs for training and inference is therefore a highly relevant goal. In this paper we present two compatible methods towards this end goal: (i) a method for introducing sparsity in the connection patterns of NNs, and (ii) a flexible hardware architecture that is compatible with training and inference-only operation and supports the proposed sparse NNs. Our approach to sparsifying a NN is extremely simple and results in a large reduction in storage and computational complexity both in training and inference modes. Moreover, this method is not tied to the hardware acceleration and provides the same benefits for training and inference in software under the current paradigm. The hardware architecture is massively parallel, but not tightly coupled to a specific NN architecture (*i.e.,* not tied to the number of nodes in a layer). Instead, the architecture allows for maximum throughput for a given amount of circuit resources.

Our approach to making a NN sparse is to specify a sparse set of neuron connections prior to training and to hold this pattern fixed throughout training and inference. We refer to this method of simply excluding some fixed set of connections in the NN as *pre-defined sparsity*. There are several methods in the literature related to sparse NNs, but most do not reduce the computation and storage complexity associated with training, which is a primary goal of this work. One related concept is dropout [10] where selected edges in the NN are not processed during some steps of the training process, but the final result is a Fully Connected (FC) NN for inference. Another set of approaches target producing a sparse NN for inference, but use FC NNs during training. Among these are pruning and trimming methods that post-process the trained NN to produce a sparse NN for inference mode [11]–[13]. As mentioned before, other methods have been proposed for reducing the complexity of performing inference on a trained FC NN such as quantization, compression, and grouping parameters [6]–[9]. Other research has suggested a method of learning sparsity during training that begins training a FC NN and uses a cost regularizer that promotes sparsity in the trained model [14]. Note that all of these methods do not substantially reduce the complexity of training and instead target inference models that have lower complexity. One method aimed at reducing both training and inference complexity is using NNs with structured, but not sparse, weight matrices [15], [16]. Finally, we note that several authors have very recently proposed pre-defined sparse NNs [17]–[19] independently of our published work [20]–[22].

Motivated by the fact that specialized hardware is typically faster and more energy efficient than GPUs and CPUs, there

Manuscript received December 3, 2018; revised February 24, 2019; accepted March 27, 2019. Date of publication April 12, 2019; date of current version June 11, 2019. This work was supported by NSF, Software and Hardware Foundations, under Grant 1763747. This paper was recommended by Guest Editor B. Murmann. *(Corresponding author: Sourya Dey.)*

The authors are with the Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089 USA (e-mail: souryade@usc.edu; kuanwenh@usc.edu; pabeerel@usc.edu; chugg@usc.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JETCAS.2019.2910864

exists a large body of literature in NN hardware acceleration. The vast majority of this addresses only inference given a trained model [9], [23]–[26], with few addressing hardware accelerated training [27]. The work of [27], for example, targets a specific size NN – *i.e.,* the logic and memory architecture is tied to the number of neurons in a layer.

We propose an architecture that supports training, but can be simplified for inference-only mode, and is flexible to the NN size. This is particularly attractive for Field Programmable Gate Array (FPGA) implementations. Specifically, the proposed architecture produces the maximum throughput on a given FPGA for a given NN and can therefore support various sized NNs on various sized FPGAs. This is accomplished by an *edge-based* processing architecture that can process z edges in a given layer in parallel (*i.e.,* we refer to z as the *degree of parallelism*). A given FPGA can support some largest value of z, and NNs with more edges will simply take more clock cycles to process.[1]

Our edge-based architecture is inspired by architectures proposed for iterative decoding of modern sparse-graph-based error correction codes (*i.e.,* Turbo and Low Density Parity Check (LDPC) codes) (cf., [28], [29]). In particular, for a given processing task, there are z logic units to perform the task and z memories to store the quantities associated with the task. A challenge with this architecture, shared between the decoding and NN applications, is that in order to achieve high throughput without memory duplication, the parallel memories must be accessed in two manners: natural order and interleaved order. In natural order, each computation unit is associated with one memory and accesses the elements of that memory sequentially. For interleaved order access, the z computational units must access the memories such that no memory is accessed more than once in a cycle. Such an addressing pattern is called *clash-free*, and this property ensures that no memory contention occurs so that no stalls or wait states are required. For modern codes, the clash-free property of the memories is ensured by defining clash-free interleavers (*i.e.,* permutations) [30], or clash-free parity check matrices [29]. In the context of NNs, this clash-free property is tied to the connection patterns between layers of neurons.

In addition to z degrees of parallelism in edge processing in a given layer, our architecture is pipelined across layers. Thus, there is a degree of parallelism associated with each layer (*i.e.,* z_i for layer i) selected to set the number of cycles required to process a layer to a constant – *i.e.,* larger layers have larger z so that the computation time of all layers is the same. For an $(L+1)$-layer NN there are L pipeline stages so that a given NN input is processed in the time it takes to complete the processing of the edges in a single layer. Furthermore, the three operations associated with training – Feedforward (FF), Backpropagation (BP), and Update of trainable parameters (UP) – are performed in parallel. The architecture may be simplified to perform only inference by eliminating the logic and memory associated with BP and UP. Furthermore, while the architecture supports the reduced complexity sparse NNs, it is also compatible with traditional FC networks. Interestingly, very recent work proposed pipelining across layers for an inference-only accelerator [31], as well as scalable edge-based architectures for training [32], [33] independently of our published work [20], [21]. Neither of these other recent works, however, takes advantage of pre-defined sparsity in the network.

Our form of pre-defined sparse connection patterns are constrained in the sense that they are designed to be compatible with our hardware architecture. We demonstrate that such hardware-friendly patterns provide learning performance better than randomly connected sparse patterns. We also compare against other state-of-the-art algorithms for generating sparse patterns which are not compatible with our hardware, and demonstrate that our patterns achieve performance statistically on-par with these. Thus, pre-defining sparsity for hardware compatibility does not cost learning performance.

A natural point to investigate is the methodology of searching for a pre-defined sparse connection pattern, which is a hyperparameter to be chosen prior to training. Accordingly, this work contains a set of trends or design guidelines for accelerating this search, resulting from detailed simulation studies of pre-defined sparsity on four different datasets.

The paper is structured as follows. Section II provides motivation for and simple examples of the effectiveness of pre-defined sparsity. Section III describes the hardware architecture in detail, including defining a class of simple clash-free connection patterns. Section IV discusses guidelines for choosing pre-defined sparse network configurations. Section V compares our method of sparsity with others. Finally, Section VI concludes the paper.

II. Structured Pre-Defined Sparsity

A. Definitions, Notation, and Background

An $(L+1)$-layer Multilayer Perceptron (MLP) has N_i nodes in the i^{th} layer, described collectively by the *neuronal configuration* $\boldsymbol{N}_{\text{net}} = (N_0, N_1, \cdots, N_L)$, where layer 0 is the input layer. We use the convention that layer i is to the 'right' of layer $i-1$. There are L *junctions* between layers, with junction i connecting the N_{i-1} nodes of its left layer $i-1$ with the N_i nodes of its right layer i.

We define pre-defined sparsity as simply not having all $N_{i-1}N_i$ edges present in junction i. Furthermore, we define *structured* pre-defined sparsity so that for a given junction i, each node in its left layer has fixed out-degree – *i.e.,* d_i^{out} connections to its right layer, and each node in its right layer has fixed in-degree – *i.e.,* d_i^{in} connections from its left layer. FC NNs have $d_i^{\text{out}} = N_i$ and $d_i^{\text{in}} = N_{i-1}$ with $N_{i-1}N_i$ edges in the i^{th} junction, while a sparse NN has at least one junction with less than this number of edges. The number of edges (or weights) in junction i is given by $|\boldsymbol{W}_i| = N_{i-1}d_i^{\text{out}} = N_i d_i^{\text{in}}$. The density of junction i is measured relative to FC and denoted as $\rho_i = |\boldsymbol{W}_i|/(N_{i-1}N_i)$. The structured constraint implies that the number of possible ρ_i values is equal to the Greatest Common Divisor (gcd) of N_{i-1} and N_i,

[1] We use the terms the terms 'connection' and 'edge' interchangeably, as we do with 'node' and 'neuron'. Also, the term 'cycle' will mean 'clock cycle', unless otherwise stated.

as shown in Appendix A. The overall density of the NN is:

$$\rho_{\text{net}} = \frac{\sum_{i=1}^{L} |\boldsymbol{W}_i|}{\sum_{i=1}^{L} N_{i-1} N_i} \tag{1}$$

Thus, specifying $\boldsymbol{N}_{\text{net}}$ and the *out-degree configuration* $\boldsymbol{d}_{\text{net}}^{\text{out}} = (d_1^{\text{out}}, \cdots, d_L^{\text{out}})$ determines the density of each junction and the overall density.

We will also consider *random pre-defined sparsity*, where connections are distributed randomly given preset ρ_i values without constraints on in- and out-degrees. In Sec. IV-B we show that random pre-defined sparsity is undesirable at low densities because it may result in unconnected neurons.

The standard equations for FC NNs are well-known [34]. For a NN using structured pre-defined sparsity, only the weights corresponding to connected edges are stored in memory and used in computation. This leads to the modified equations (2)–(4), where subscripts denote layer/junction numbers, single superscripts denote neurons in a layer, and double superscripts denote (right neuron, left neuron) in a junction. The FF processing proceeds left-to-right and computes the activations $\boldsymbol{a}_i$ and associated derivatives $\dot{\boldsymbol{a}}_i$ for each layer by applying an activation function act$(\cdot)$ to a linear combination of biases $\boldsymbol{b}_i$, junction weights $\boldsymbol{W}_i$, and preceding layer activations $\boldsymbol{a}_{i-1}$:

$$h_i^{(j)} = \sum_{f=1}^{d_i^{\text{in}}} W_i^{(j,k_f)} a_{i-1}^{(k_f)} + b_i^{(j)} \tag{2a}$$

$$a_i^{(j)} = \text{act}\left(h_i^{(j)}\right) \tag{2b}$$

$$\dot{a}_i^{(j)} = \frac{da_i^{(j)}}{dh_i^{(j)}} = \dot{\text{act}}\left(h_i^{(j)}\right) \tag{2c}$$

Note that (2c) is used in training, but is not required in inference mode. The BP computation is done only in training and computes a sequence of error values from right-to-left:

$$\delta_L^{(j)} = \frac{\partial l^{(j)}\left(a_L^{(j)}, y^{(j)}\right)}{\partial h_L^{(j)}} \tag{3a}$$

$$\delta_i^{(j)} = \dot{a}_i^{(j)} \left(\sum_{f=1}^{d_i^{\text{out}}} W_{i+1}^{(k_f,j)} \delta_{i+1}^{(k_f)} \right) \tag{3b}$$

where $l^{(j)}\left(a_L^{(j)}, y^{(j)}\right)$ is the j^{th} component of the loss function. Finally, stochastic gradient UP is given by

$$b_i^{(j)} \leftarrow b_i^{(j)} - \eta \delta_i^{(j)} \tag{4a}$$

$$W_i^{(j,k)} \leftarrow W_i^{(j,k)} - \eta a_{i-1}^{(k)} \delta_i^{(j)} \tag{4b}$$

where η is the learning rate. The parameters on the left-hand-side of (2)–(4) will be referred to as the *network parameters*, with the weights and biases being the *trainable parameters*.

B. Motivation and Preliminary Examples

Pre-defined sparsity can be motivated by inspecting the histogram for trained weights in a FC NN. There have been previous efforts to study such statistics [3], [35], however, not for individual junctions. Fig. 1 shows weight histograms for each junction in both a 2-junction and 4-junction FC NN

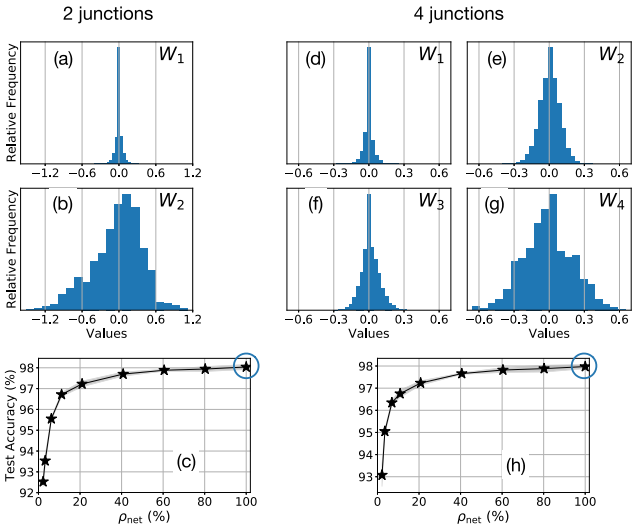

Fig. 1. Histograms of weight values in different junctions for FC NNs trained on MNIST for 50 epochs, with (a-b) $\boldsymbol{N}_{\text{net}} = (800, 100, 10)$, and (d-g) $\boldsymbol{N}_{\text{net}} = (800, 100, 100, 100, 10)$. Test accuracy shown in (c,h) for different NNs with same $\boldsymbol{N}_{\text{net}}$ and varying ρ_{net}. ρ_{net} is set by reducing ρ_1 since junction 1 has more weights close to zero in the FC cases (circled).

trained on the MNIST dataset. Note that many of the weights are zero or near-zero after training, especially in the earlier junctions. This motivates the idea that some weights in these layers could be set to zero (*i.e.,* the edges excluded). Even with this intuition, it is unclear that one can pre-define a set of weights to be zero and let the NN learn around this pre-defined sparsity constraint. Fig. 1(c) and (h) show that, in fact, this is the case – *i.e.,* this shows classification accuracy as a function of the overall density ρ_{net} for structured pre-defined sparsity. Since the computational and storage complexity is directly proportional to the number of edges in the NN, operating at an overall density of, for example, 50% results in a 2X reduction in complexity both during training and inference. Detailed numerical experiments in Section IV build on these simple examples. However, before we proceed to those results, it is important to consider a hardware architecture that can support structured pre-defined sparsity and consider the additional clash-free constraints placed on the connection patterns so that these can be considered in the studies in Section IV.

III. Hardware Architecture

In this section we describe the proposed flexible hardware architecture outlined in the Introduction. The overall architectural view is captured by Fig. 2: sub-figure (a) shows parallel edge processing within a junction with degree of parallelism 3, (b) shows clash-free memory access, and (c) junction pipelining and parallel processing of the three operations – FF, BP, UP. The toy example in Fig. 2(a)-(b) is for $N_{i-1} = 6$, $N_i = 3$, $\rho_i = 6/18 = 1/3$, and $z_i = 3$. Fig. 2(a) shows that the $z_i = 3$ blue edges are processed in parallel in one cycle, while the pink edges are processed in parallel during the next cycle. Fig. 2(b) shows how the $z_i = 3$ FF processing logic units access the memories in natural and interleaved order. As will be described in detail in Sec. III-B, the interleaved order access may represent reading of the activations $\{a_{i-1}^{(j)}\}$ for $j \in \{0, 1, 5\}$ and the natural order access may correspond

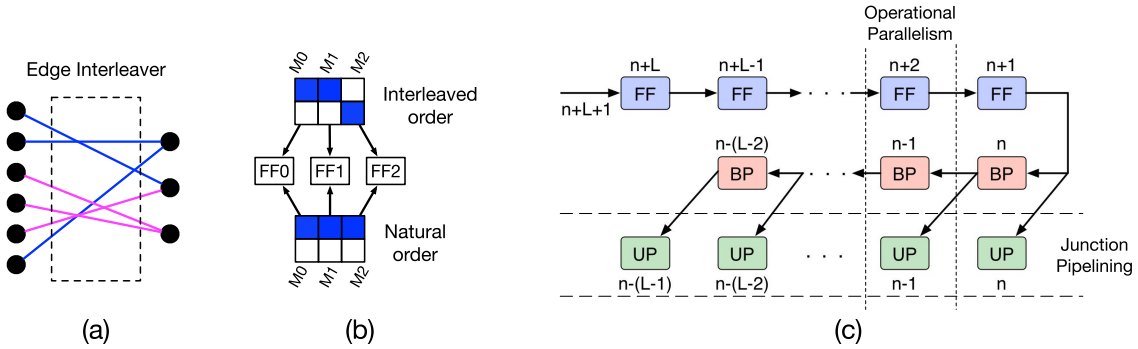

Fig. 2. (a) Processing $z_i = 3$ edges in each cycle (blue in cycle 0, pink in cycle 1) for some junction i . (b) Accessing $z_i = 3$ memories – M0, M1 and M2 shown as columns – from two separate banks, one in natural order (same address from each memory), the other in interleaved order. Clash-freedom is achieved by accessing only one element from each memory. The accessed values are fed to $z_i = 3$ processors to perform FF simultaneously. (c) Operational parallelism in each junction (vertical dotted lines denote processing for one junction), and junction pipelining of each operation across junctions (horizontal dashed lines) in a multi-junction NN. Subfigure (c) is modified from our previous conference publication [20, Fig. 2(c)].

to writing the computed activations $\{a_i^{(j)}\}$ for $j \in \{0, 1, 2\}$. On the next cycle, the remaining memory locations (*i.e.,* the white cells) will be accessed. Note that this illustrates a clash-free connection pattern since each of the $z_i = 3$ memories is accessed no more than once in each cycle – *i.e.,* one hit per column on each access.

The junction-based operation in Fig. 2(b) is repeated for each junction in a pipeline. In particular, there are L pipeline stages. For example, for the FF pipeline, while the first stage is processing input vector $n + L$ on junction 1, the second stage is processing input vector $n + L - 1$ on junction 2. The degree of parallelism for each junction is selected so that the processing time for any operation (FF/BP/UP) is the same for each junction. Thus the throughput, *i.e.,* the frequency of processing input samples, is determined by the time taken to perform a single operation in a single junction.

In summary, the architecture is (i) edge-based and not tied to a specific number of nodes in a layer, (ii) flexible in that the amount of logic is determined by the degree of parallelism which trades size for speed, and (iii) fully pipelined for the parallel operations associated with NN training. Also note that the architecture can be specialized to perform only inference by removing the logic and memory associated with the BP and UP operations, and the $\dot{\boldsymbol{a}}_i$ computation in (2c).

A key concern when implementing NNs on hardware is the large amount of storage required. Several characteristics regarding memory requirements guided us in developing the proposed architecture. Firstly, since weight memories are the largest, their number should be minimized. Secondly, having a few deep memories is more efficient in terms of power and area than having many shallow memories [36]. Thirdly, throughput should be maximized without duplicating memories, hence the need for clash-free connection patterns.

In Sec. III-A, we describe junction pipelining design, which attempts to minimize weight storage resources. The memory organization within a junction is described in Sec. III-B, and is designed to minimize the number of memories for a given degree of parallelism. Finally, clash-free access conditions are developed in Sec. III-B and III-C, and a simple method for implementing such patterns given in Sec. III-C.

A. Junction Pipelining and Operational Parallelism

Our edge-based architecture is motivated by the fact that all three operations – FF, BP, UP – use the same weight values for computation. Since z_i edges are processed in parallel in a single cycle, the time taken to complete an operation in junction i is $C_i = |\boldsymbol{W}_i| / z_i$ cycles. The *degree of parallelism configuration* $\boldsymbol{z}_{\text{net}} = (z_1, \cdots, z_L)$ is chosen to achieve $C_i = C \quad \forall i \quad \in \{1, \cdots, L\}$. This allows efficient junction pipelining since each operation takes exactly C cycles to be completed for each input in each junction, which we refer to as a *junction cycle*.[2] This determines throughput.

The following is an analysis of Fig. 2(c) in more detail for an example NN with $L = 2$. While a new training input numbered $n+3$ is getting loaded as $\boldsymbol{a}_0$, junction 1 is processing the FF stage for the previous input $n + 2$ and computing $\boldsymbol{a}_1$. Simultaneously, junction 2 is processing FF and computing cost $\boldsymbol{\delta}_L$ via cost derivatives for input $n + 1$. It is also doing BP on input n to compute $\boldsymbol{\delta}_1$, as well as updating (UP) its parameters from the finished $\boldsymbol{\delta}_L$ computation of input n. Simultaneously, junction 1 is performing UP using $\boldsymbol{\delta}_1$ from the finished BP results of input $n - 1$. This results in *operational parallelism* in each junction, as shown in Fig. 3. The combined speedup is approximately a factor of $3L$ as compared to doing one operation at a time for a single input.

Notice from Fig. 3 that there is only one weight memory bank which is accessed for all three operations. However, UP in junction 1 needs access to $\boldsymbol{a}_0$ for input $n - 1$, as per the weight update equation (4b). This means that there need to be $2L + 1 = 5$ left activation memory banks for storing $\boldsymbol{a}_0$ for inputs $n - 1$ to $n + 3$, *i.e.,* a queue-like structure. Similarly, UP in junction 2 will need $2(L - 1) + 1 = 3$ queued banks for each of its left activation $\boldsymbol{a}_1$ and its derivative $\dot{\boldsymbol{a}}_1$ memories – for inputs from n (for which values will be read) to $n + 2$ (for which values are being computed and written). There also need to be 2 banks for all $\boldsymbol{\delta}$ memories –

[2]During hardware implementation, a few extra cycles may be needed to flush the pipeline, so $C_i = |\boldsymbol{W}_i| / z_i + c_i$. These are also balanced, *i.e.,* $c_i = c \quad \forall i \in \{1, \cdots, L\}$, to achieve efficient pipelining. In our initial implementation [37], for example, $c = 2$, and the junction cycle is $C = 34$.

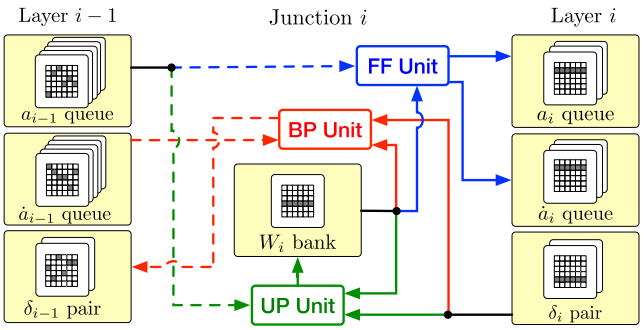

Fig. 3. Architecture for parallel operations for an intermediate junction i ($i \neq 1, L$) showing the three operations along with associated inputs and outputs. Natural and interleaved order accesses are shown using solid and dashed lines, respectively. The $\boldsymbol{a}$ and $\dot{\boldsymbol{a}}$ memory banks occur as queues, the $\boldsymbol{\delta}$ memory banks as pairs, while there is a single weight memory bank. Figure modified from our previous conference publication [20, Fig. 3].

TABLE I

HARDWARE ARCHITECTURE TOTAL STORAGE COST COMPARISON FOR $\boldsymbol{N}_{\text{net}} = (800, 100, 10)$ FC VS SPARSE WITH $\boldsymbol{d}_{\text{net}}^{\text{out}} = (20, 10)$, $\rho_{\text{net}} = 21\%$

Parameter	Expression	Count (FC)	Count (sparse)
$\boldsymbol{a}$	$\sum_{i=0}^{L-1} (2(L-i)+1)\, N_i$	4300	4300
$\dot{\boldsymbol{a}}$	$\sum_{i=1}^{L-1} (2(L-i)+1)\, N_i$	300	300
$\boldsymbol{\delta}$	$2\sum_{i=1}^{L} N_i$	220	220
$\boldsymbol{b}$	$\sum_{i=1}^{L} N_i$	110	110
$\boldsymbol{W}$	$\sum_{i=1}^{L} N_i d_i^{\text{in}}$	**81000**	**17000**
TOTAL	Σ (All above)	85930	21930

one for reading and the other for writing. Thus junction pipelining requires multiple memory banks, but only for layer parameters $\boldsymbol{a}$, $\dot{\boldsymbol{a}}$ and $\boldsymbol{\delta}$, *not* for weights.[3] The number of layer parameters is insignificant compared to the number of weights for practical networks. This is why pre-defined sparsity leads to significant storage savings, as quantified in Table I for the circled FC point vs. the $\rho_{\text{net}} = 21\%$ point from Fig. 1(c). Specifically, memory requirements are reduced by 3.9X in this case. Furthermore, the computational complexity, which is proportional to the number of weights for a MLP, is reduced by 4.8X. For this example, these complexity reductions come at a cost of degrading the classification accuracy from 98.0% to 97.2%.

B. Memory Organization

For the purposes of memory organization, edges are numbered sequentially from top to bottom on the right side of the junction. Other network parameters such as $\boldsymbol{a}$, $\dot{\boldsymbol{a}}$ and $\boldsymbol{\delta}$ are numbered according to the neuron numbers in their respective layer. Consider Fig. 4 as an example, where junction i is flanked by $N_{i-1} = 12$ left neurons with $d_i^{\text{out}} = 2$ and $N_i = 8$ right neurons, leading to $|\boldsymbol{W}_i| = 24$ and $d_i^{\text{in}} = 3$. The three weights connecting to right neuron 0 are numbered 0, 1, 2; the next three connecting to right neuron 1 are numbered 3, 4, 5, and so on. A particular right neuron connects to some subset of left neurons of cardinality d_i^{in}.

[3]This is achieved by making the weight memory dual-port, while $\boldsymbol{a}$ and $\dot{\boldsymbol{a}}$ are single-ported memories. The $\boldsymbol{\delta}$ memories are also dual-ported due to the exact manner in which we implemented this architecture on FPGA, refer to [37] for full details.

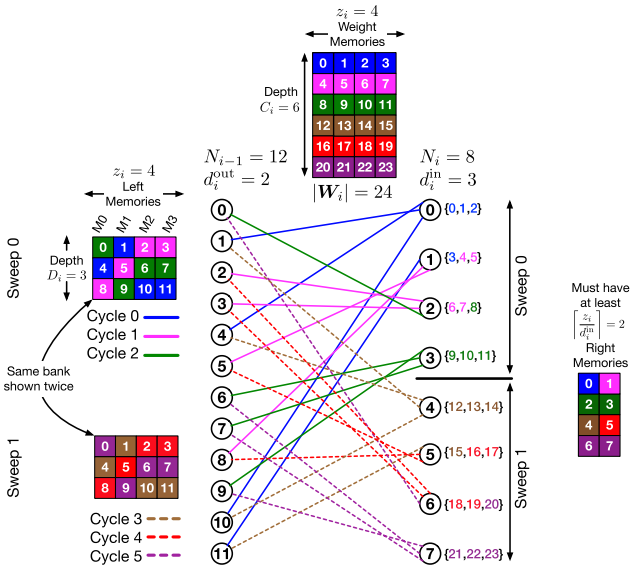

Fig. 4. An example of processing inside junction i with $z_i = 4$ memories in the weight and left banks, and $z_{i+1} = 2$ memories in the right bank. The banks are represented as numerical grids, each column is a memory, and the number in each cell is the number of the edge / left neuron / right neuron whose parameter value is stored in it. Edge are sequentially numbered on the right (shown in curly braces). Four weights are read in each of the six cycles with the first three colored blue, pink and green, respectively. These represent sweep 0, while the next 3 (using dashed lines) colored brown, red and purple, respectively, represent sweep 1. Clash-freedom leads to at most one cell from each memory in each bank being accessed each cycle. Weight and right memories are accessed in natural order, while left memories are accessed in interleaved order.

Each type of network parameter is stored in a bank of memories. The example in Fig. 4 uses $z_i = 4$, *i.e.,* 4 weights are accessed per cycle. We designed the weight memory bank to have the minimum number of memories to prevent clashes, *i.e.,* z_i, and their depth equals C_i. Weight memories are read in natural order – one row per cycle (shown in same color). This implies that the logic to compute memory addresses simply consists of z_i incrementers.

Right neurons are processed sequentially due to the weight numbering. The number of right neuron parameters of a particular type needing to be accessed in a cycle is upper bounded by $\lceil z_i/d_i^{\text{in}} \rceil$, which leads to $z_{i+1} \geq \lceil z_i/d_i^{\text{in}} \rceil$ in order to prevent clashes in the right memory bank.[4] For FF in Fig. 4 for example, cycles 0 and 1 finish computation of $a_i^{(0)}$ and $a_i^{(1)}$ respectively, while cycle 2 finishes computing both $a_i^{(2)}$ and $a_i^{(3)}$. For BP or UP, everything remains same except for the right memory accesses. Now $\delta_i^{(0)}$ and $\delta_i^{(1)}$ are used in cycle 0, $\delta_i^{(1)}$ and $\delta_i^{(2)}$ in cycle 1, and $\delta_i^{(2)}$ and $\delta_i^{(3)}$ in cycle 2. Thus the maximum number of right neuron parameters ever accessed in a cycle is $\lceil z_i/d_i^{\text{in}} \rceil = 2$.

Since edges are interleaved on the left, in general, the z_i edge processing logic units will need access to z_i parameters of a particular type from layer $i-1$. So all the left memory banks have z_i memories, each of depth $D_i = N_{i-1}/z_i$, which are accessed in interleaved order. For example, after D_i cycles, N_{i-1} edges have been processed – *i.e.,* $(D_i \times z_i) = N_{i-1}$.

[4]This does not limit most practical designs (see Appendix B).

We require that each of these edges be connected to a different left neuron to eliminate the possibility of duplicate edges. This completes a *sweep*, *i.e.,* one complete access of the left memory bank. Since each left neuron connects to d_i^{out} edges, d_i^{out} sweeps are required to process all the edges, *i.e.,* each left activation is read d_i^{out} times in the whole junction cycle. The reader can verify that D_i cycles multiplied by d_i^{out} sweeps results in C_i total cycles, *i.e.,* one junction cycle.

C. Clash-Free Connection Patterns

We define a *clash* as attempting to perform a particular operation more than once on the same memory at the same time, which would stall processing.[5] The idea of *clash-freedom* is to pre-define a pattern of connections and z values such that no operation in any junction of the NN results in a clash. Sec. III-B described how z values should be designed to prevent clashes in the weight and right memory banks.

This subsection analyzes the left memory banks, which are accessed in interleaved order. Their memory access pattern should be designed so as to prevent clashes. Additionally, the following properties are desired for practical clash-free patterns. Firstly, it should be easy to find a pattern that gives good performance. Secondly, the logic and storage required to generate the left memory addresses should be low complexity.

We generate clash-free patterns by initially specifying the left memory addresses to be accessed in cycle 0 using a seed vector $\boldsymbol{\phi}_i \in \{0, 1, \cdots, D_i - 1\}^{z_i}$. Subsequent addresses are cyclically generated. Considering Fig. 4 as an example, $\boldsymbol{\phi}_i = (1, 0, 2, 2)$. Thus in cycle 0, we access addresses $(1, 0, 2, 2)$ from memories $(M0, M1, M2, M3)$, *i.e.,* left neurons $(4, 1, 10, 11)$. In cycle 1, the accessed addresses are $(\boldsymbol{\phi}_i + 1) \,\%\, D_i = (2, 1, 0, 0)$, and so on. Since $D_i = 3$, cycles 3–5 access the same left neurons as cycles 0–2.

We found that this technique results in a large number of possible connection patterns, as discussed in Appendix C. Randomly sampling from this set results in performance comparable with non-clash-free NNs, as shown in Sec. IV-B. Finally, our approach only requires storing $\boldsymbol{\phi}_i$ and using z_i incrementers to generate subsequent addresses. This approach is similar to methods used in modern coding to allow parallel processing and memory accesses, c.f. [28]–[30]. Appendix C discusses other techniques to generate clash-free patterns.

D. Batch Size

It is common in training of NNs to use minibatches. For a batch size of M, the UP operation in (4) is performed only once for M inputs by using the average over the M gradients. Our architecture performs an UP for every input and therefore may be viewed as having batch size one. However, the processing in our architecture differs from a typical software implementation with $M = 1$ due to the pipelined and parallel operations. Specifically, in our architecture, FF and BP for the same input use different weights, as implied by Fig. 2(c). In results not presented here, we found no performance degradation due to this variation from the standard backpropagation algorithm. There is considerable ambiguity in the literature regarding ideal batch sizes [38], [39], and we found that our current network architecture performed well in our initial hardware implementation [37].

The architecture can be modified by performing UP once every $(M > 1)$ samples. As an example, a practical case for a deep MLP could be $M = 64$ and $L = 5$. For all such cases where $M \gg L$, FF and BP will use the same weights for most inputs in a batch.

Finally, a more conventional minibatch update can also be obtained by completely removing the UP logic from the junction pipeline. After performing FF and BP on M samples, the pipeline will be flushed and the averaged gradients over M samples used to update all parameters in all junctions simultaneously.

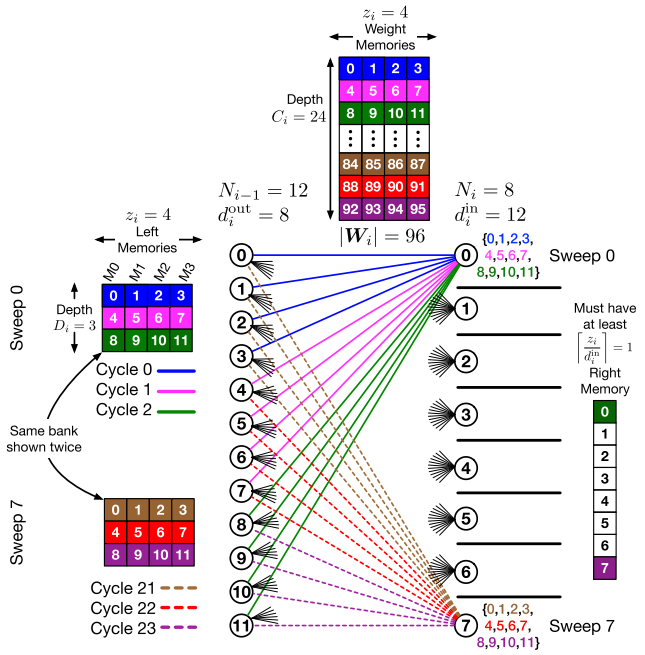

Fig. 5. Processing the FC version of the junction from Fig. 4. For clarity, only the first 12 and last 12 edges (dashed) are shown, corresponding respectively to right neurons 0 and 7, sweeps 0 and 7, cycles 0–2 and 21–23.

E. Special Case: Processing a FC Junction

Fig. 5 shows the FC version of the junction from Fig. 4, which has 96 edges to be accessed and operated on. This can be done keeping the same junction cycle $C_i = 6$ by increasing z_i to 16, *i.e.,* using more hardware. On the other hand, if hardware resources are limited, one can use the same $z_i = 4$ and pay the price of a longer junction cycle $C_i = 24$, as shown in Fig. 5. This demonstrates the flexibility of our architecture.

Note that FC junctions are clash-free in all practical cases due to the following reasons. Firstly, the left memory accesses are in natural order just like the weights, which ensures that no more than one element is accessed from each memory per cycle. Secondly, $\lceil z_i / d_i^{\text{in}} \rceil = 1$ for all practical cases since

[5]For single-ported memories, attempting two reads or two writes or a read and a write in the same cycle is a clash. For simple dual-ported memories with one port exclusively for reading and the other exclusively for writing, a read and a write can be performed in the same cycle. Attempting to perform two reads or two writes in the same cycle is a clash.

$z_i \leq N_{i-1}$, as discussed in Appendix B, and $d_i^{\text{in}} = N_{i-1}$ for FC junctions. This means that at most one right neuron is processed in a cycle, so clashes will never occur when accessing the right memory bank.

Note that compared to Fig. 4, the weight memories in Fig. 5 are deeper since C_i has increased from 6 to 24. However, the left layer memories remain the same size since $N_{i-1} = 12$ and $z_i = 4$ are unchanged, but the left memory bank is accessed more times since the number of sweeps has increased from 2 to 8. Also note that even if cycle 0 (blue) accesses some other clash-free subset of left neurons, such as $(4, 5, 6, 7)$ instead of $(0, 1, 2, 3)$, the connection pattern would remain unchanged. This implies that different memory access patterns do not necessarily lead to different connection patterns; as discussed further in Appendix C.

IV. Observed Trends of Pre-Defined Sparsity

This section analyzes trends observed when experimenting with several different classification datasets via software simulations. We intend the following four trends to provide guidelines on designing pre-defined sparse NNs.

1) Hardware-compatible, clash-free, pre-defined sparse patterns perform at least as well as other pre-defined sparse patterns (*i.e.,* random and structured) (Sec. IV-B).
2) The performance of pre-defined sparsity is better on datasets that have more inherent redundancy (Sec. IV-C).
3) Junction density should increase to the right: junctions closer to the output should generally have more connections than junctions closer to the input (Sec. IV-D).
4) Larger and more sparse NNs are better than smaller and denser NNs, given the same number of layers and trainable parameters. Specifically, 'larger' refers to more hidden neurons (Sec. IV-E).

The remainder of this section first describes the datasets we experimented on, and then examines these trends in detail.

A. Datasets and Experimental Configuration

Unless otherwise noted, the following parameters and configurations listed below were used for all presented results.

a) MNIST handwritten digits [40]: We rasterized each input image into a single layer of 784 features,[6] *i.e.,* the permutation-invariant format. No data augmentation was applied.

b) Reuters RCV1 corpus of newswire articles [41]: The classification categories are grouped in a tree structure. We used preprocessing techniques similar to [42] to isolate articles which fell under a single category at the second level of the tree. We finally obtained 328,669 articles in 50 categories, split into 50,000 for validation, 100,000 for test, and the remaining for training. The original data has a list of token strings for each story, for example, a story on finance would frequently contain the token 'financ'. We chose the most common 2000 tokens and computed counts for each of these in each article. Each count x was transformed into $\log(1 + x)$ to form the final 2000-dimensional feature vector for each input.

c) TIMIT speech corpus [43]: TIMIT is a speech dataset comprising approximately 5.4 hours of 16 kHz audio commonly used in Automatic Speech Recognition (ASR). A modern ASR system has three major components: (i) preprocessing and feature extraction, (ii) acoustic model, and (iii) dictionary and language model. A complete study of an ASR system is beyond the scope of this work. Instead we focus on the acoustic model, which is typically implemented using a NN. The input to the acoustic model is feature vectors and the output is a probability distribution on phonemes (*i.e.,* speech sounds). For our experiments, we used 25ms speech frames with 10ms shift, as in [42], and computed a feature vector of 39 Mel-frequency Cepstral Coefficient (MFCC)s for each frame. We used the complete training set of 818,837 training samples (462 speakers), 89,319 validation samples (50 speakers), and 212,093 test samples (118 speakers). We used a phoneme set of size 39 as defined in [44].

d) CIFAR-100 images [45]: Our setup for CIFAR-100 consists of a Convolutional Neural Network (CNN) followed by a MLP. The CNN has 3 blocks and each block has 2 convolutional layers with window size 3x3 followed by a max pooling layer of pool size 2x2. The number of filters for the six convolutional layers is (60,60, 125,125, 250,250). This results in a total of approximately one million trainable parameters in the convolutional portion of the network. Batch normalization is applied before activations. The output from the 3rd block, after flattening into a vector, has 4000 features. Typically dropout is applied in the MLP portion, however we omitted it there since pre-defined sparsity is an alternate form of parameter reduction. Instead we found that a dropout probability of half applied to the convolutional blocks improved performance. No data augmentation was applied.

For each dataset, we performed classification using one-hot labels and measured accuracy on the test set as a performance metric. We also calculated the top-5 test set classification accuracy for CIFAR-100.

We found the optimal training configuration for each FC setup by doing a grid search using validation performance as a metric. This resulted in choosing ReLU activations for all layers except for the final softmax layer. The initialization proposed by He *et al.* [46] worked best for the weights; while for biases, we found that an initial value of 0.1 worked best in all cases except for Reuters, for which zeroes worked better. The Adam optimizer [47] was used with all parameters set to default, except that we set the decay parameter to 10^{-5} for best results. We used a batch size of 1024 for TIMIT and Reuters since the number of training samples is large, and 256 for MNIST and CIFAR.

All experiments were run for 50 epochs of training and regularization was applied as an L2 penalty to the weights. To maintain consistency, we kept most hyperparameters the same when sparsifying the network, but reduced the L2 penalty coefficient with increasing sparsity. This was done because sparse NNs have fewer trainable parameters and are less prone to overfitting. We ran each experiment at least five times to

[6]On certain occasions we added 16 input features which are always trivially 0 so as to get 800 features for each input. This leads to easier selection of different sparse network configurations. In Fig. 1(c) for example, we used 800 input neurons and 100 hidden neurons since $\gcd(800, 100) > \gcd(784, 100)$, so more values of ρ_{net} can be simulated.

TABLE II
COMPARISON OF PRE-DEFINED SPARSE METHODS

$d_{\text{net}}^{\text{out}}$	$\rho_{\text{net}}\%$	z_{net}	Test Accuracy Performance		
			Clash-free	Structured	Random
MNIST: $N_{\text{net}} = (800, 100, 100, 100, 10)$, FC test accuracy = $(98 \pm 0.1)\%$					
$(80, 80, 80, 10)$	80.2	$(200, 25, 25, 4)$	97.9 ± 0.2	97.9 ± 0.2	97.8 ± 0.2
$(60, 60, 60, 10)$	60.4	$(200, 25, 25, 4)$	97.6 ± 0.1	97.8 ± 0.1	97.6 ± 0.2
$(40, 40, 40, 10)$	40.6	$(200, 25, 25, 5)$	97.5 ± 0.1	97.7	97.6 ± 0.1
$(20, 20, 20, 10)$	20.8	$(200, 25, 25, 10)$	97.2 ± 0.2	97.2 ± 0.1	97.1 ± 0.1
$(10, 10, 10, 10)$	10.9	$(200, 25, 25, 25)$	96.7 ± 0.1	96.8 ± 0.2	96.7 ± 0.2
$(5, 10, 10, 10)$	6.9	$(100, 25, 25, 25)$	96.3 ± 0.1	96.3 ± 0.1	96.2 ± 0.1
$(2, 5, 5, 10)$	3.6	$(80, 25, 25, 50)$	95 ± 0.2	95.1 ± 0.1	95 ± 0.3
$(1, 2, 2, 10)$	2.2	$(80, 20, 20, 100)$	93.3 ± 0.3	93.1 ± 0.5	92 ± 0.3
Reuters: $N_{\text{net}} = (2000, 50, 50)$, FC test accuracy = $(89.6 \pm 0.1)\%$					
$(25, 25)$	50	$(1000, 25)$	89.4 ± 0.1	89.3	89.4
$(10, 10)$	20	$(400, 10)$	87 ± 0.1	86.7 ± 0.1	86.5 ± 0.1
$(5, 5)$	10	$(200, 5)$	78.5 ± 0.5	78.2 ± 0.7	77.5 ± 0.6
$(2, 2)$	4	$(80, 2)$	53.3 ± 1.8	51.2 ± 1.7	46.8 ± 2.9
$(1, 1)$	2	$(40, 1)$	28.4 ± 2.4	28.7 ± 2.3	28 ± 1.9
TIMIT: $N_{\text{net}} = (39, 390, 39)$, FC test accuracy = $(43.2 \pm 0.2)\%$					
$(270, 27)$	69.2	$(13, 13)$	43 ± 0.1	43	43 ± 0.1
$(180, 18)$	46.2		42.7 ± 0.1	42.8 ± 0.1	42.9 ± 0.1
$(90, 9)$	23.1		42.1 ± 0.1	42.5 ± 0.1	42.4 ± 0.1
$(60, 6)$	15.4		41.5 ± 0.1	41.8 ± 0.2	41.9 ± 0.1
$(30, 3)$	7.7		40.5 ± 0.2	40.1 ± 0.2	39.4 ± 0.8
CIFAR-100 [a]: $N_{\text{net}} = (4000, 500, 100)$, FC top-5 test accuracy = $(87.1 \pm 0.6)\%$					
$(100, 100)$	22	$(2000, 250)$	87.5 ± 0.2	87.7 ± 0.2	87.4 ± 0.3
$(29, 29)$	6.4		86.8 ± 0.3	87.2 ± 0.5	87.1 ± 0.2
$(12, 12)$	2.6	$(400, 50)$	86.3 ± 0.2	86.5 ± 0.4	86.6 ± 0.4
$(5, 5)$	1.1		85.3 ± 0.5	85.5 ± 0.5	85.7 ± 0.3
$(2, 2)$	0.4	$(80, 10)$	84.1 ± 0.5	84.3 ± 0.3	83.8 ± 0.3
$(1, 1)$	0.2		83 ± 0.5	83.3 ± 0.4	81.7 ± 0.7

[a]For CIFAR-100, given values of N_{net}, $d_{\text{net}}^{\text{out}}$, z_{net} and ρ_{net} are just for the MLP portion, which follows a CNN as described in Sec. IV-A to form the complete net. Reported values are top-5 test accuracies obtained from training on the complete net.

average out randomness and we show the 90% Confidence Interval (CI)s for each metric as shaded regions (this also holds for the results in Fig. 1(c,h)). In addition to the results shown, we developed a data set of Morse code symbol sequences and investigated pre-defined sparse NNs. While these results are excluded for brevity, they are consistent with the trends described in this Section, and can be found in [48].

B. Comparison of Pre-Defined Sparse Methods

Table II shows performance on different datasets for three methods of pre-defined sparsity: a) the most restrictive and hardware-friendly clash-freedom, b) structured, and c) random. For the clash-free case, we experimented with different z_{net} settings to simulate different hardware environments:

- Reuters: One junction cycle is 50 cycles for all the different densities. This is because we scale z_{net} accordingly, *i.e.,* a more powerful hardware device is used for each NN as ρ_{net} increases.
- CIFAR-100 and MNIST: These simulate cases where hardware choice is limited, such as a high-end, a mid-range and a low-end device being available. Thus three different z_{net} values are used for CIFAR-100 depending on ρ_{net}.
- TIMIT: We keep z_{net} constant for different densities. Junction cycle length varies from 90 cycles for $\rho_{\text{net}} = 7.69\%$ to 810 for $\rho_{\text{net}} = 69.23\%$. This shows that when limited to a single low-end hardware device, denser NNs can be processed in longer time by simply changing z_{net}.

Table II confirms that *hardware-friendly clash-free pre-defined sparse architectures do not lead to any statistically significant performance degradation*. We also observed that random pre-defined sparsity performs poorly for very low density networks, as shown by the blue values. This is possibly because there is non-negligible probability of neurons getting completely disconnected, leading to irrecoverable loss of information.

C. Dataset Redundancy

Many machine learning datasets have considerable redundancy in their input features. For example, one may not need information from the ∼800 input features of MNIST to infer the correct image class. We hypothesize that pre-defined sparsity takes advantage of this redundancy, and will be less effective when the redundancy is reduced. To test this, we changed the feature vector for each dataset as follows. For MNIST, Principal Component Analysis (PCA) was used to reduce the feature count to the least redundant 200. For Reuters, the number of most frequent tokens considered as features was reduced from 2000 to 400. For TIMIT, we both reduced and increased the number of MFCCs by 3X to 13 and 117, respectively. Note that the latter increases redundancy. For CIFAR-100, a source of redundancy is the depth of the CNN, which extracts features and discriminates between classes before the MLP performs final classification. In other words, the CNN eases the burden of the MLP. So a way to reduce redundancy and increase the classification burden of the MLP is to lessen the effectiveness of the CNN by reducing its depth. Accordingly, we used a single convolutional layer with 250 filters of window size 5×5, followed by a 8×8 max pooling layer. This results in the same number of features, 4000, at the input of the MLP as the original network, but has reduced redundancy for the MLP.

Classification performance results are shown in Fig. 6 as a function of ρ_{net}. For MNIST and CIFAR-100, the performance degrades more sharply with reducing ρ_{net} for the nets using the reduced redundancy datasets. To explore this further, we recreated the histograms from Fig. 1 for the reduced redundancy datasets, *i.e.,* a FC NN with $N_{\text{net}} = (200, 100, 10)$ training on MNIST after PCA. We observed a wider spread of weight values, implying less opportunity for sparsification (*i.e.,* fewer weights were close to zero). Similar trends are less discernible for Reuters and TIMIT, however, reducing redundancy led to worse performance overall.

The results in Fig. 6 further demonstrate the effectiveness of pre-defined sparsity in greatly reducing network complexity with negligible performance degradation. For example, even the reduced redundancy problems perform well when operating with half the number of connections. For CIFAR in particular, *FC performs worse than an overall MLP density of around 20%*. Thus, in addition to reducing complexity, structured pre-defined sparsity may be viewed as an alternative to dropout in the MLP for the purpose of improving classification.

D. Individual Junction Densities

The weight histograms in Fig. 1 indicate that latter junctions, particularly junction L closest to the output, have a wide spread of weight values. This suggests that a good strategy

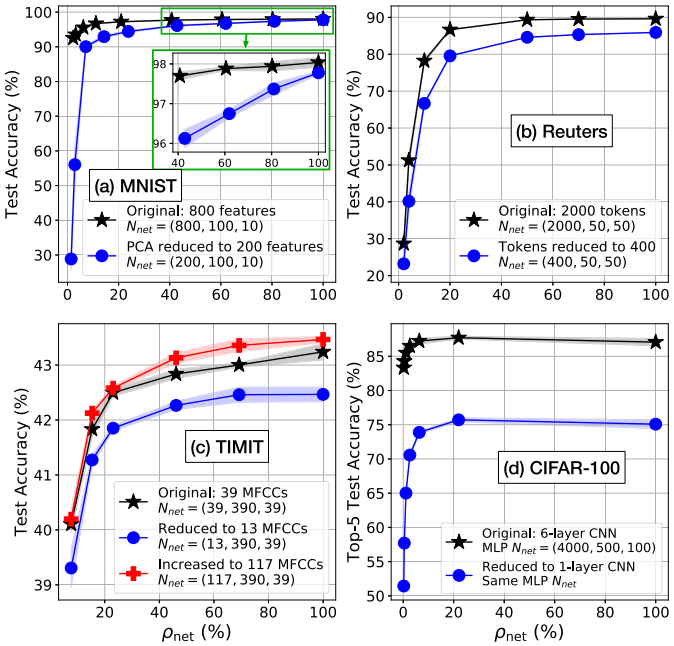

Fig. 6. Comparison of classification accuracy as a function of ρ_{net} for different versions of datasets – original, reduced in redundancy by reducing feature space (MNIST, Reuters, TIMIT) or performing less processing prior to the MLP (CIFAR-100), and increasing redundancy by enlarging feature space (TIMIT). Higher density points for MNIST are magnified.

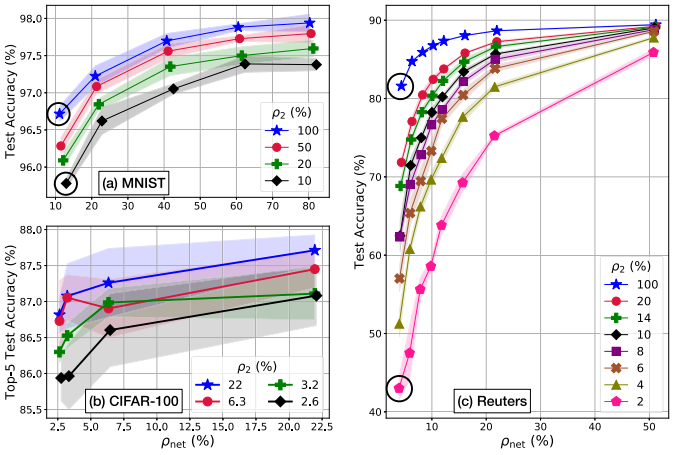

Fig. 7. Comparison of classification accuracy as a function of ρ_{net} for different ρ_L, where $L = 2$. Black-circled points show the effects of ρ_2 when ρ_{net} is the same. $\boldsymbol{N}_{\text{net}}$ values are (800, 100, 10) for MNIST, (2000, 50, 50) for Reuters, and (4000, 500, 100) for the MLP in CIFAR-100.

for reducing ρ_{net} would be to use lower densities in earlier junctions – *i.e.,* $\rho_1 < \rho_L$. This is demonstrated in Fig. 7 for the cases of MNIST, CIFAR-100 and Reuters, each with $L = 2$ junctions in their MLPs. Each curve in each subfigure is for a fixed ρ_2, *i.e.,* reducing ρ_{net} across a curve is done solely by reducing ρ_1. For a fixed ρ_{net}, the performance improves as ρ_2 increases. For example, the circled points in Reuters both have $\rho_{\text{net}} = 4\%$, but the starred point with $\rho_2 = 100\%$ has approximately 40% better test accuracy than the pentagonal point with $\rho_2 = 2\%$. The trend clearly holds for MNIST and Reuters, and is also discernible for CIFAR-100. We observed a similar trend for three-junction NNs, *i.e.,* $\rho_3 > \rho_2$ improves performance, but the detailed results are omitted for brevity.

We further observed that this trend is related to the redundancy inherent in the dataset and may not hold for datasets

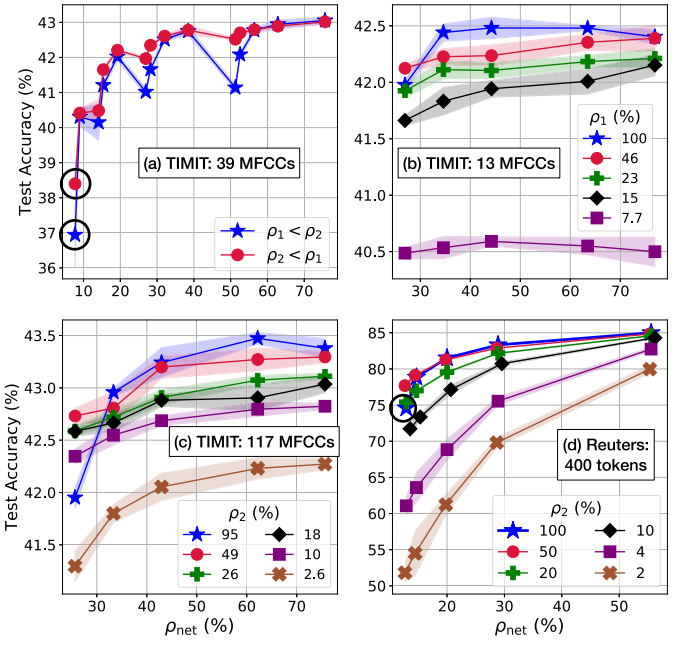

Fig. 8. Comparison of classification accuracy as a function of ρ_{net} for: (a) TIMIT with 39 MFCCs for the two cases where one junction is always sparser than the other and vice-versa. Black-circled points show how reducing ρ_1 degrades performance to a greater extent. (b) TIMIT with 13 MFCCs for different ρ_1. (c,d) TIMIT with 117 MFCCs, and Reuters reduced to 400 tokens, for different ρ_2. $\boldsymbol{N}_{\text{net}}$ values are (a) (39, 390, 39), (b) (13, 390, 39), (c) (117, 390, 39), (d) (400, 50, 50).

with very low levels of redundancy. To explore this, results analogous to those in Fig. 7 are presented in Fig. 8 for TIMIT, but with varying sized MFCC feature vectors – *i.e.,* datasets corresponding to larger feature vectors will contain more redundancy. The results in Fig. 8(c) are for 117 dimensional MFCCs and are consistent with the trend in Fig. 7. However, for a MFCC dimension of 13, this trend actually reverses – *i.e.,* junction 1 should have higher density. This is shown in Fig. 8(b), where each curve is for a fixed ρ_1. This reversed trend is also observed for the case of 39 dimensional feature vectors, considered in Fig. 8(a), where $\boldsymbol{N}_{\text{net}} = (39, 390, 39)$. Due to this symmetric neuronal configuration, for each value of ρ_{net} on the x-axis in Fig. 8(a), the two curves have complementary values of ρ_1 and ρ_2 ($\rho_1 \neq \rho_2$) – *e.g.,* the two curves at $\rho_{\text{net}} = 7.69\%$ have (ρ_1, ρ_2) pairs of $(2.56\%, 12.82\%)$ and $(12.82\%, 2.56\%)$. We observe that the curve for $\rho_1 < \rho_2$ is generally worse than the curve for $\rho_2 < \rho_1$, which indicates that junction 1 should have higher density in this case.

Fig. 8(d) depicts the results for Reuters with the feature vector size reduced to 400 tokens. While junction 2 is still more important (as in Fig. 7(c) for the original Reuters dataset), notice the circled star-point at the very left of the $\rho_2 = 100\%$ curve. This point has very low ρ_1. Unlike Fig. 7(c), it crosses below the other curves, indicating that it is more important to have higher density in the first junction with this less redundant set of features. We observed a similar, but less prominent, trend in MNIST PCA when the feature dimension was reduced to 200.

In summary, if an individual junction density falls below a certain value, referred to as the *critical junction density*, it will adversely affect performance regardless of the density of other junctions. This explains why some of the curves cross

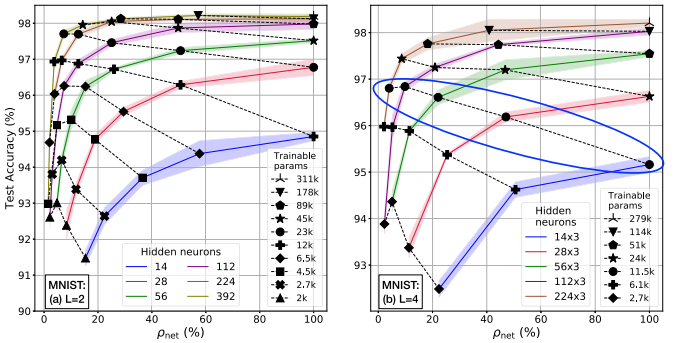

Fig. 9. Comparing 'large and sparse' to 'small and dense' networks for MNIST with 784 features, with (a) $\boldsymbol{N}_{\text{net}} = (784, x, 10)$ (on the left), and (b) $\boldsymbol{N}_{\text{net}} = (784, x, x, x, 10)$ (on the right). Solid curves (with the shaded CIs around them) are for constant x, black dashed curves with same marker are for same number of trainable parameters. The final junction is always FC. Intermediate junctions for the $L = 4$ case have d^{out} values similar to junction 1.

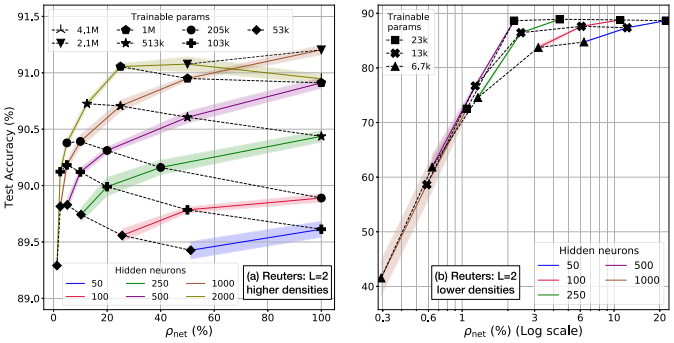

Fig. 10. Comparing 'large and sparse' to 'small and dense' networks for Reuters with 2000 tokens, with $\boldsymbol{N}_{\text{net}} = (2000, x, 50)$. The x-axis is split into higher values on the left (a), and lower values on the right in log scale (b). Solid curves (with the shaded CIs around them) are for constant x, black dashed curves with same marker are for same number of trainable parameters. Junction 1 is sparsified first until its number of total weights is approximately equal to that of junction 2, then both are sparsified equally.

in Fig. 8. The critical junction density is much smaller for earlier junctions than for later junctions in most datasets with sufficient redundancy. However, the critical density for earlier junctions increases for datasets with low redundancy.

E. 'Large and Sparse' vs. 'Small and Dense' Networks

We observed that when keeping the total number of trainable parameters the same, sparser NNs with larger hidden layers (*i.e.,* more neurons) generally performed better than denser networks with smaller hidden layers. This is true as long as the larger NN is not so sparse that individual junction densities fall below the critical density, as explained in Sec. IV-D. While the critical density is problem-dependent, it is usually low enough to obtain significant complexity savings above it. Thus, 'large and sparse' is better than 'small and dense' for many practical cases, including NNs with more than one hidden layer (*i.e.,* $L > 2$).

Fig. 9 shows this for networks having one and three hidden layers trained on MNIST. For the three layer network, all hidden layers have the same number of neurons. Each solid curve shows classification performance vs. ρ_{net} for a particular $\boldsymbol{N}_{\text{net}}$, while the black dashed curves with identical markers are configurations that have approximately the same number of trainable parameters. As an example, the points with circular markers (with a big blue ellipse around them) in Fig. 9(b) all have the same number of trainable parameters and indicate that the larger, more sparse NNs perform better. Specifically, the network with $\boldsymbol{N}_{\text{net}} = (784, 112, 112, 112, 10)$ and $\boldsymbol{d}^{\text{out}}_{\text{net}} = (10, 10, 10, 10)$ corresponding to $\rho_{\text{net}} = 9.82\%$ performs significantly better than the FC network with $\boldsymbol{N}_{\text{net}} = (784, 14, 14, 14, 10)$, and other smaller and denser networks, despite each having 11,500 trainable parameters. Increasing the network size further to $\boldsymbol{N}_{\text{net}} = (784, 224, 224, 224, 10)$, and reducing ρ_{net} to 4% to fix the number of trainable parameters at 11,500, leads to performance degradation. This is because this ρ_{net} was achieved by setting $\rho_2 = \rho_3 = 2.68\%$, which appears to be below the critical density.

Fig. 10 summarizes the analogous experiment on Reuters with similar conclusions. Both subfigures are for the same results, with the x-axis split into higher and lower density

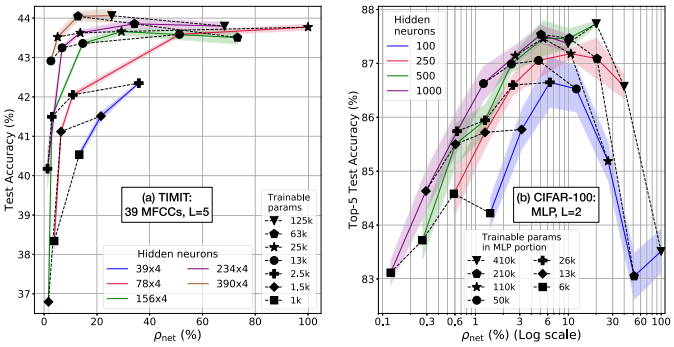

Fig. 11. Comparing 'large and sparse' to 'small and dense' networks for (a) TIMIT with 39 MFCCs and $\boldsymbol{N}_{\text{net}} = (39, x, x, x, 39)$ (on the left), and (b) CIFAR-100 with the deep 9-layer CNN and MLP $\boldsymbol{N}_{\text{net}} = (4000, x, 100)$ with log scale for the x-axis (on the right). Solid curves (with the shaded CIs around them) are for constant x, black dashed curves with same marker are for same number of trainable parameters (in the MLP portion only for CIFAR). Since TIMIT has symmetric junctions, we tried to keep input and output junction densities as close as possible and adjusted intermediate junction densities to get the desired ρ_{net}. CIFAR-100 is sparsified in a way similar to Reuters in Fig. 10.

range (on log scale) to show more detail. Observe that the trend of 'large and sparse' being better than 'small and dense' holds for subfigure (a), but reverses for (b) since densities are very low (the black dashed curves have positive slope instead of negative). This is due to the critical density effect.

Fig. 11(a) shows the result for the same experiment on TIMIT with four hidden layers.[7] The trend is less clearly discernible, but it exists. Notice how the black dashed curves have negative slopes at appreciable levels of ρ_{net}, indicating 'large and sparse' being better than 'small and dense', but high positive slopes at low ρ_{net}, indicating the rapid degradation in performance as density is reduced beyond the critical density. This is exacerbated by the fact that TIMIT with 39 MFCCs is a dataset with low redundancy, so the effects of very low ρ_{net} are better observed.

Fig. 11(b) for the MLP portion of CIFAR-100 shows similar results as TIMIT, but on a log x-scale for more clarity. As noted in Sec. IV-C, the best performance for a given $\boldsymbol{N}_{\text{net}}$

[7]We also performed experiments on TIMIT with one hidden layer ($L = 2$) and Reuters with 2 hidden layers ($L = 3$). Results were similar to those shown, hence are not included for brevity's sake.

occurs at an overall density less than 100%. It appears that for any N_{net} for CIFAR-100, peak performance occurs at around 10–20% overall MLP density. In experiments not shown here, we obtained similar results for the reduced redundancy net with a single convolutional layer.

V. Comparison to Other Sparse NN Methods

Numerical results in Sec. IV showed that hardware-compatible clash-free connection patterns performed as well as structured and random pre-defined sparse connections. In this section, we compare clash-free patterns against two sparsity approaches that are less constrained than the structured pre-defined sparsity considered in Sec. IV. In particular, both approaches remove the constraint of regular degree – *i.e.,* these approaches yield sparse NNs that have varying d_i^{out} and d_i^{in} selected to optimize classification performance.

A. Attention-Based Preprocessed Sparsity

Previous works [49], [50] have applied the concept of *attention* on object recognition and image captioning to achieve better performance with fewer parameters and less computation. We simplify this idea by computing the variance of input features as attention and setting the out-degree of the neurons of the input layer based on this value. Specifically, the feature variances are quantized into three levels, and input neurons with higher attention are assigned more connections than those with lower attention. For the neurons in latter layers, we use uniform out-degree and in-degree.

B. Learning Structured Sparsity During Training

While the method in Sec. V-A obtains a non-uniform neuron out-degree for the first layer, it only considers the properties of the dataset and not the learning process. We also compared against the method of Learning Structured Sparsity (LSS) which learns a good sparse connection pattern during training. This method was proposed in [14] and prunes the connections during training by using a sparse-promoting penalty function $p(\cdot)$ as part of the objective function (which also includes a loss function $l(\cdot)$ and a regularizer $r(\cdot)$):

$$\min_{\{\boldsymbol{W}_i,\boldsymbol{b}_i\}_{i=1}^{L}} l\left(\{\boldsymbol{W}_i,\boldsymbol{b}_i\}_{i=1}^{L}\right)+\lambda r\left(\{\boldsymbol{W}_i\}_{i=1}^{L}\right)+\sum_{i=1}^{L}\gamma_i\, p(\boldsymbol{W}_i)$$

where the penalty coefficients $\{\gamma_i\}_{i=1}^{L}$ control the density of each junction. Increasing γ_i decreases ρ_i, however, obtaining a specific value of ρ_i requires experimental tuning of γ_i. Example $p(\cdot)$ functions include L1 and L1/L2 used in Lasso [51] and group-Lasso [52], respectively. In the results presented in this section, we used L1 as the element-wise sparse-promoting penalty function and L2 as the regularizer.

Note that, in contrast to the attention-based method and the structured pre-defined sparsity approach, LSS is not a pre-defined sparsity method. Instead training in LSS begins with a FC network, which means that training complexity is similar to that of a FC NN. At the end of the LSS training process, weights with absolute value below a threshold are set as zero to achieve the target density.

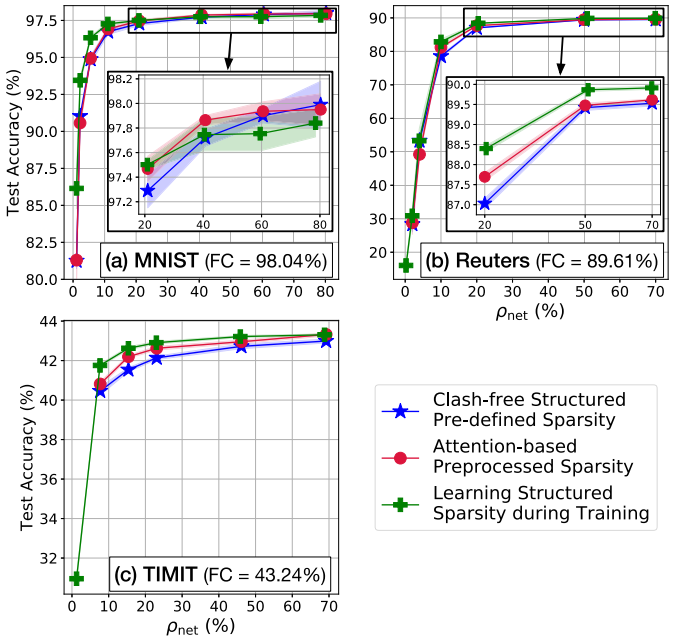

Fig. 12. Comparison of classification accuracy as a function of ρ_{net} for different sparse methods on (a) MNIST with $N_{\text{net}} = (800, 100, 10)$, (b) Reuters with $N_{\text{net}} = (2000, 50, 50)$, and (c) TIMIT with $N_{\text{net}} = (39, 390, 39)$. We set the overall density ρ_{net} and all individual junction densities ρ_i to be approximately the same across different sparse methods. The FC NN is the same for each approach, and its test accuracy performance is also noted.

C. Performance Comparison

Fig. 12 compares performance versus ρ_{net} of different sparse NNs on MNIST, Reuters, and TIMIT. The individual density of each junction with the attention-based preprocessed sparse method is set to be identical to the density of each junction using clash-free pre-defined sparse method. However, the density of the nets using the LSS method can be tuned only with the penalty coefficients. We tuned these to approximate match the density of the other methods.

The LSS method performs best among all sparse methods, which is to be expected as it is the least constrained and also discovers a good sparse connection pattern during training. However, the performance with clash-free pre-defined sparsity is near that of the attention-based and LSS methods – *i.e.,* within 2% in terms of test accuracy at $\rho_{\text{net}} = 20\%$. We conclude that even though the clash-free patterns are highly structured and pre-defined, there is no significant performance degradation when compared to advanced methods for producing sparse models which exploit specific properties of the dataset or learn sparse patterns during training. In fact, the performance of our method is comparable to FC NNs at most values of ρ_{net}.

VI. Conclusions and Future Work

In this work we proposed a new technique for complexity reduction of neural networks – pre-defined sparsity – in which a fixed sparse connection pattern is enforced prior to training and held fixed during both training and inference. We presented a hardware architecture suited to leverage the benefits of structured pre-defined sparsity, capable of parallel and pipelined processing. The architecture can be used for

both training and inference modes, and supports networks of arbitrary density, including conventional fully connected ones. Flexibility is afforded by the degree of parallelism z_{net}, which trades hardware complexity for speed. Simple methods for clash-free memory access are presented and shown to achieve performance on par with the best known methods for obtaining sparse MLPs.

Using extensive numerical experiments, we also identified trends and guidelines which help in designing pre-defined sparse networks. Firstly, it is better to allocate connections in a structured manner rather than randomly. Secondly, for most datasets with high redundancy, earlier junctions can be made more sparse. Thirdly, it is better to have more neurons in the hidden layers, and then sparsify aggressively to keep the number of edges low and reduce complexity. Interesting areas for future research include analytical approaches to justify the trends observed in this work.

This work focuses on MLPs, which are either used standalone, or as building blocks in a wide variety of problems such as classification and ASR systems. There are other kinds of NNs such as convolutional and recurrent. The former is characterized by weight reuse and local connectivity, and will likely need a different parallel memory access scheme to avoid clashes. The latter undergoes backpropagation through time and can have potential hardware bottlenecks due to state updates. We believe the methods introduced in this work also have applicability to convolutional and recurrent architectures, however, these represent a significant extension of our methods and are excellent topics for future research.

While our hardware implementation in [37] is a good initial proof-of-concept, there are significant engineering tasks required to demonstrate state-of-the-art training speeds. This is an area of ongoing work.

In conclusion, the rapidly growing complexity associated with modern NNs is a major challenge. Pre-defined sparsity is a simple method to help address this challenge, as is acceleration with custom hardware. Speeding the training process by orders of magnitude would allow more extensive search over NN architectures, and therefore a better understanding of the largely empirical process of NN design.

Appendix A
Structured Pre-Defined Sparsity Constraints

In our structured pre-defined sparse network, ρ_i, the density of junction i, cannot be arbitrary, since $\rho_i = d_i^{\text{out}}/N_i = d_i^{\text{in}}/N_{i-1}$, where d_i^{out} and d_i^{in} are natural numbers satisfying the equation $N_{i-1}d_i^{\text{out}} = N_i d_i^{\text{in}}$. Therefore, the number of possible ρ_i values is the same as the number of $\left(d_i^{\text{out}}, d_i^{\text{in}}\right)$ values satisfying the structured pre-defined sparsity constraints:

$$d_i^{\text{out}} = \frac{N_i d_i^{\text{in}}}{N_{i-1}}, \quad d_i^{\text{in}} \leq N_{i-1},\ d_i^{\text{out}},\ d_i^{\text{in}} \in \mathbb{N} \tag{5}$$

where $\mathbb{N}$ denotes the set of natural numbers.

The smallest value of d_i^{in} which satisfies $d_i^{\text{out}} \in \mathbb{N}$ is $N_{i-1}/\text{gcd}(N_{i-1}, N_i)$, and other values are its integer multiples. Since d_i^{in} is upper bounded by N_{i-1}, the total number of possible $\left(d_i^{\text{out}}, d_i^{\text{in}}\right)$ is $\text{gcd}(N_{i-1}, N_i)$. Thus, the set of possible ρ_i is

$$\left\{\rho_i \in (0,1] \,\middle|\, \rho_i = \frac{k}{\text{gcd}(N_{i-1}, N_i)}, k \in \mathbb{N}\right\}. \tag{6}$$

Appendix B
Hardware Architecture Constraints

The depth of left memories in our hardware architecture is $D_i = N_{i-1}/z_i$. Thus, N_{i-1} should preferably be an integral multiple of z_i. This is not a burdening constraint since the choice of z_i is independent of network parameters and depends on the capacity of the device. In the unusual case that this constraint cannot be met, the extra cells in memories can be filled with dummy values such as 0.

There are also 2 conditions placed on the z values to eliminate stalls in processing: for all junctions $i \in \{1, \cdots, L\}$, (i) $|\boldsymbol{W}_i|/z_i = C$, and (ii) $z_{i+1} \geq \lceil z_i/d_i^{\text{in}} \rceil$ (excluding $i = L$). Using the definitions from Sec. II-A, (i) is equivalent to $z_{i+1} = z_i d_{i+1}^{\text{out}}/d_i^{\text{in}}$. Then, (ii) can be equivalently written as:

$$d_{i+1}^{\text{out}} \geq \frac{d_i^{\text{in}}}{z_i}\left\lceil \frac{z_i}{d_i^{\text{in}}} \right\rceil \tag{7}$$

which needs to be satisfied $\forall\ i \in \{1, \cdots, L-1\}$. In practice, it is desirable to design z_i/d_i^{in} to be an integer so that an integral number of right neurons finish processing every cycle. This simplifies hardware implementation by eliminating the need for additional storage, for example, of the intermediate activation values during FF. In this case, (7) reduces to $d_{i+1}^{\text{out}} \geq 1$, which is always true.

For non-integral z_i/d_i^{in}, there are two cases. If $z_i > d_i^{\text{in}}$, (7) reduces to $d_{i+1}^{\text{out}} \geq 2$. On the other hand, if $z_i < d_i^{\text{in}}$, there is no bound on the right hand side of (7). In general, note that (7) becomes a burdening constraint only if d_i^{in} is large, and d_{i+1}^{out} and z_i are both desired to be small. This corresponds to earlier junctions being denser than later, which is typically not desirable according to the observations in Sec. IV-D, or to very limited hardware resources. We thus conclude that (7) is not a limiting constraint in most practical cases.

Appendix C
Clash-Free Patterns

Specifying N_{i-1}, N_i, d_i^{in} and z_i for junction i in a clash-free structured pre-defined sparse NN does not uniquely define a connection pattern (unless it is FC). This section discusses the *number of possible left memory access patterns* S_{M_i} for such a junction i. Note that the total number of possible memory access patterns for the complete NN is $S_M = \prod_{i=1}^{L} S_{M_i}$.

When $z_i \geq d_i^{\text{in}}$, which is expected to be true for practical cases of implementing sparse NNs on powerful hardware devices, S_{M_i} is also equal to the *number of possible connection patterns* S_{C_i}, which is the key quantity of interest. This is because if $z_i \geq d_i^{\text{in}}$, at least one right neuron is completely processed in some cycle. Thus, changing the left memory access pattern will change the left neurons to which that right neuron connects, thereby changing the connection pattern. This one-to-one correspondence results in $S_{M_i} = S_{C_i}$.

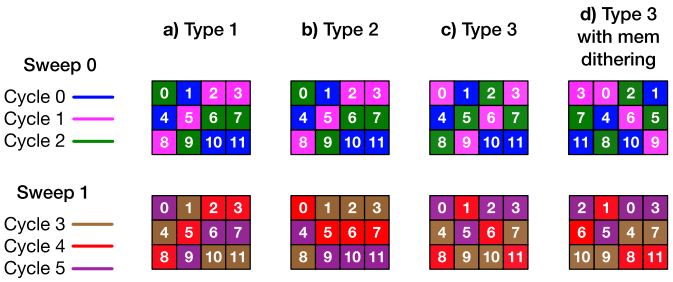

Fig. 13. (a-c) Various types of clash-freedom, and (d) memory dithering for type 3, using the same left neuronal structure from Fig. 4 as an example. The grids represent different access patterns for the same memory bank. The number in each cell represents the left neuron number whose parameter is stored in that cell. Cells sharing the same color are read in the same cycle.

For the case of $z_i < d_i^{\text{in}}$, a FC junction provides an example where $S_{M_i} \neq S_{C_i}$. Specifically, in this case $S_{C_i} = 1$ as there is only one way to fully connect all neurons, but there are many clash-free memory access patterns. This is shown next, along with various types of clash-freedom.

Type 1 is as described in Sec. III-C, and recapitulated in Fig. 13(a). The number of ways of designing $\boldsymbol{\phi}_i$ is $S_{M_i} = {D_i}^{z_i}$.

In *Type 2*, implemented in our earlier work [37], a new $\boldsymbol{\phi}_i$ is defined for every sweep. Considering the example in Fig. 13(b), $\boldsymbol{\phi}_i = (1, 0, 2, 2)$ for sweep 0, but $(2, 0, 0, 0)$ for sweep 1. There will be d_i^{out} different $\boldsymbol{\phi}_i$ vectors for each junction, resulting in $S_{M_i} = {D_i}^{z_i d_i^{\text{out}}}$.

In *Type 3*, the constraint of cyclically accessing the left memories is also eliminated. Instead, any cycle can access any cell from each of the memories. This means that storing $\boldsymbol{\phi}_i$ is not enough, the entire sequence of memory accesses needs to be stored as a matrix $\boldsymbol{\Phi}_i \in \{0, 1, \cdots, D_i - 1\}^{D_i \times z_i}$. This removes the need of having z_i incrementers to compute subsequent addresses. In Fig. 13(c) for example, $\boldsymbol{\Phi}_i = ((1, 0, 2, 2), (0, 2, 1, 0), (2, 1, 0, 1))$ for sweep 0. Every sweep would also have a different $\boldsymbol{\Phi}_i$, resulting in $S_{M_i} = (D_i!)^{z_i d_i^{\text{out}}}$.

A technique that can be applied to all the types of clash-freedom is *memory dithering*, which is a permutation of the z_i memories (*i.e.,* the columns) in a bank. This permutation can change every sweep, as shown in Fig. 13(d). Memory dithering incurs an additional address computation storage cost because of the z_i permutation, but increases S_{M_i} by a factor K_i. If d_i^{in}/z_i is an integer, an integral number of cycles are required to process each right neuron. Since a cycle accesses all memories, dithering has no effect and $K_i = 1$. On the other hand, if z_i/d_i^{in} is an integer greater than 1, the effects of dithering on connectivity patterns are only observed when switching from one right neuron to the next within a cycle. This results in: $K_i = \left(\frac{z_i!}{\frac{z_i}{d_i^{\text{in}}}! {d_i^{\text{in}}}^{\frac{z_i}{d_i^{\text{in}}}}} \right)^{d_i^{\text{out}}}$ for types 2 and 3, and the d_i^{out} exponent is omitted for type 1 since the access pattern does not change across sweeps.

When either of z_i or d_i^{in} does not perfectly divide the other, an exact value of K_i is hard to arrive at since some proper or improper fraction of right neurons are processed every cycle. In such cases, K_i is upper-bounded by $(z!)^{d_i^{\text{out}}}$.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[2] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with COTS HPC systems," in *Proc. Int. Conf. Mach. Learn. (ICML)*, vol. 28, 2013, pp. III-1337–III-1345.

[3] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 1135–1143.

[4] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[5] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[6] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev. (2014). "Compressing deep convolutional networks using vector quantization." [Online]. Available: https://arxiv.org/abs/1412.6115

[7] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 1–10.

[8] S. Han, H. Mao, and W. J. Dally. (2015). "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." [Online]. Available: https://arxiv.org/abs/1510.00149

[9] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[11] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 1–13.

[12] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 267–278.

[13] A. Aghasi, A. Abdi, N. Nguyen, and J. Romberg, "Net-trim: Convex pruning of deep neural networks with performance guarantee," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 3177–3186.

[14] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2016, pp. 2074–2082.

[15] V. Sindhwani, T. Sainath, and S. Kumar, "Structured transforms for small-footprint deep learning," in *Proc. Adv. Neural Inform. Process. Syst. (NIPS)*, 2015, pp. 3088–3096.

[16] S. Wang *et al.*, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2018, pp. 11–20.

[17] A. Bourely, J. P. Boueri, and K. Choromonski. (2017). "Sparse neural networks topologies." [Online]. Available: https://arxiv.org/abs/1706.05683

[18] A. Prabhu, G. Varma, and A. M. Namboodiri. (2017). "Deep expander networks: Efficient deep networks from graph theory." [Online]. Available: https://arxiv.org/abs/1711.08757

[19] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," *Nature Commun.*, vol. 9, Jun. 2018, Art. no. 2383.

[20] S. Dey, Y. Shao, K. M. Chugg, and P. A. Beerel, "Accelerating training of deep neural networks via sparse edge processing," in *Proc. 26th Int. Conf. Artif. Neural Netw. (ICANN)*. Cham, Switzerland: Springer, Sep. 2017, pp. 273–280.

[21] S. Dey, P. A. Beerei, and K. M. Chugg, "Interleaver design for deep neural networks," in *Proc. 51st Asilomar Conf. Signals, Syst., Comput.*, Oct./Nov. 2017, pp. 1979–1983.

[22] S. Dey, K.-W. Huang, P. A. Beerel, and K. M. Chugg, "Characterizing sparse connectivity patterns in neural networks," in *Proc. Inf. Theory Appl. Workshop (ITA)*, Feb. 2018, pp. 1–9.

[23] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[24] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-S. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration*, vol. 62, pp. 14–23, Jun. 2018.
[25] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016, Art. no. 20.
[26] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2016, pp. 16–25.
[27] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, 2014, pp. 269–284.
[28] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 3, pp. 369–379, Sep. 1999.
[29] T. Brack *et al.*, "Low complexity LDPC code decoders for next generation standards," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Apr. 2007, pp. 1–6.
[30] S. Crozier and P. Guinand, "High-performance low-memory interleaver banks for turbo-codes," in *Proc. IEEE 54th Veh. Technol. Conf.*, vol. 4, Oct. 2001, pp. 2394–2398.
[31] F. Sun *et al.*, "A high-performance accelerator for large-scale convolutional neural networks," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. IEEE Int. Conf. Ubiquitous Comput. Commun. (ISPA/IUCC)*, Dec. 2017, pp. 622–629.
[32] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 3, pp. 513–517, Mar. 2017.
[33] T. Guan, X. Zeng, and M. Seok. (2017). "Recursive binary neural network learning model with 2.28b/weight storage requirement." [Online]. Available: https://arxiv.org/abs/1709.05306
[34] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org
[35] J. Yosinski and H. Lipson, "Visually debugging restricted Boltzmann machine training with a 3D example," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2012, pp. 1–6.
[36] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. London, U.K.: Pearson, 2010.
[37] S. Dey *et al.*, "A highly parallel FPGA implementation of sparse neural network training," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs (ReConFig)*, Dec. 2018, pp. 1–4. [Online]. Available: https://arxiv.org/abs/1806.01087
[38] P. Goyal *et al.* (2017). "Accurate, large minibatch SGD: Training imagenet in 1 hour." [Online]. Available: https://arxiv.org/abs/1706.02677
[39] D. Masters and C. Luschi. (2018). "Revisiting small batch training for deep neural networks." [Online]. Available: https://arxiv.org/abs/1804.07612
[40] Y. LeCun, C. Cortes, and C. J. Burges. *The MNIST Database of Handwritten Digits*. Accessed: Jul. 1, 2018. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[41] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "RCV1: A new benchmark collection for text categorization research," *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, Dec. 2004.
[42] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. (2012). "Improving neural networks by preventing co-adaptation of feature detectors." [Online]. Available: https://arxiv.org/abs/1207.0580
[43] J. S. Garofolo *et al. TIMIT Acoustic-Phonetic Continuous Speech Corpus*. Accessed: Aug. 28, 2018. [Online]. Available: https://catalog.ldc.upenn.edu/LDC93S1
[44] K.-F. Lee and H.-W. Hon, "Speaker-independent phone recognition using hidden Markov models," *IEEE Trans. Acoust., Speech Signal Process.*, vol. 37, no. 11, pp. 1641–1648, Nov. 1989.
[45] A. Krizhevsky, "Learning multiple layers of features from tiny images," M.S. thesis, Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, 2009.
[46] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1026–1034.
[47] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2014.
[48] S. Dey, K. M. Chugg, and P. A. Beerel, "Morse code datasets for machine learning," in *Proc. 9th Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT)*, Jul. 2018, pp. 1–7.
[49] J. Ba, V. Mnih, and K. Kavukcuoglu. (2014). "Multiple object recognition with visual attention." [Online]. Available: https://arxiv.org/abs/1412.7755
[50] K. Xu *et al.*, "Show, attend and tell: Neural image caption generation with visual attention," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 2048–2057.
[51] M. R. Osborne, B. Presnell, and B. A. Turlach, "A new approach to variable selection in least squares problems," *IMA J. Numer. Anal.*, vol. 20, no. 3, pp. 389–403, 2000.
[52] R. Jenatton, J.-Y. Audibert, and F. Bach, "Structured variable selection with sparsity-inducing norms," *J. Mach. Learn. Res.*, vol. 12, pp. 2777–2824, Feb. 2011.

Sourya Dey received the B. Tech. degree in instrumentation engineering from IIT Kharagpur, India, in 2014. He is currently pursuing the Ph.D. degree in electrical engineering with the University of Southern California, Los Angeles, CA, USA. His research focuses on sparsity, model search, and algorithm-hardware co-design of neural networks in machine learning.

Kuan-Wen Huang received the B.S. degree in electrical engineering from National Taiwan University, Taipei, in 2012. He is currently pursuing the Ph.D. degree in electrical engineering with the University of Southern California. His current research focuses primarily on sparse and low-rank signal processing, optimization, and deep learning.

Peter A. Beerel received the B.S.E. degree in electrical engineering from Princeton University, Princeton, NJ, USA, in 1989, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1991 and 1994, respectively. He is currently a Full Professor and an Associate Chair of the Computer Engineering Division, Ming Hsieh Electrical and Computer Engineering Department, University of Southern California. He co-founded TimeLess Design Automation to commercialize an asynchronous ASIC flow, in 2008, and sold the company, in 2010, to Fulcrum Microsystems which was bought by Intel, in 2011. His interests include a variety of topics in CAD, VLSI, and machine learning.

Keith M. Chugg (S'88–M'95–SM'06–F'10) received the B.S. degree (Hons.) in engineering from the Harvey Mudd College, Claremont, CA, USA, in 1989, and the Ph.D. degree in electrical engineering from the University of Southern California (USC), Los Angeles, CA, USA, in 1995. Since 1996, he has been on the faculty of the Ming Hsieh Department of Electrical and Computer Engineering, USC, where he is currently a Professor. He is a co-founder of TrellisWare Technologies, Inc., where he serves as the Chief Scientist. His research interests are in the general areas of signal processing, digital communications, machine learning, and associated efficient implementations.