DENNI: Distributed Neural Network Inference on Severely Resource Constrained Edge Devices

Rohit Sahu*, Ryan Toepfer*, Mathew D. Sinclair†, and Henry Duwe*
* Iowa State University, Ames, IA † University of Wisconsin-Madison & AMD Research, Madison, WI
rsahu@iastate.edu, ryantpfr@gmail.com, sinclair@cs.wisc.edu, duwe@iastate.edu

***Abstract*—Pervasive intelligence promises to revolutionize society from Industrial Internet of Things (IIoT), to smart infrastructure and homes, to personal health monitoring. Unfortunately, many edge devices that are pervasively embedded into infrastructure or implanted into humans are severely resource-constrained. As performing computations at the edge becomes increasingly important to meet latency deadlines and retain sensitive data locally, severe resource constraints present a challenge because many algorithms are too large to fit on a single edge device. In this paper, we focus on distributing inference for neural networks (NNs) with convolution and fully connected layers across multiple edge nodes. In order to improve efficiency on severely resource-constrained edge nodes for diverse NN architectures we present an end-to-end, automated approach, DENNI, that optimizes NN distribution with minimal nodes while meeting memory constraints. When targeting a network of edge nodes with 256KB of non-volatile memory connected with Bluetooth Low Energy, DENNI successfully distributes NN inference for a variety of machine learning algorithms across multiple edge nodes where other, static approaches cannot.**

***Index Terms*—distributed computing, neural networks, wireless sensor nodes**

I. Introduction

Smart IoT edge devices that make intelligent decisions autonomously in real-time, adapt to the environment, and intelligently collaborate with each other have potential to revolutionize everything from manufacturing to transportation to personal healthcare [1]. However, unpredictable connection to gateways, limited network bandwidth, limited energy capacity of extreme edge devices, and privacy concerns with uploading private end-user information to the cloud for analysis have driven intelligence to be moved to the edge [2]. Given that there are expected to be 43 billion IoT edge devices by 2023 [3] and 43% of all machine learning (ML) inference will be performed completely on the edge devices [4], these issues will be significantly exacerbated.

As IoT devices become increasingly pervasive, collectively they can provide a significant amount of computation. However, these devices are severely resource-constrained. For example, as shown in Table I some of the most frequently used IoT processors only have a few hundred KB's of memory and frequencies less than 100 MHz. Such device characteristics are fundamentally at odds with the requirements of most neural networks (NNs), which can have hundreds of MBs of parameters (or more) [5]. These constraints make it extremely difficult to perform end-to-end inference on a single edge device. Given these constraints, some prior work redesigns NNs to fit on a single device, often utilizing compression, pruning, and quantization to trade off accuracy for footprint [6]–[16] or offloading computations to the cloud [17]. However, even after applying these techniques many state-of-the-art convolutional neural networks (CNNs) still require more memory than a single edge device possesses [5]. Therefore, it is critical to develop methods to distribute NN inference on a set of resource-constrained edge devices.

Prior works have also examined distributing NN inference in systems with larger memory capacities (e.g., GBs per node) or high performance processors [18]–[25] These approaches use some variation of early exiting or layerwise splitting, which works well on devices with larger memories. However, their approaches also require at least one layer to be executed on a single device, which is not always possible on severely resource-constrained edge devices such as those listed in Table I. Other recent work examines distributing NN inference to minimize for overall inference latency and communication [26], [27], while others examines distributing NN inference to smaller, more resource constrained devices like Raspberry Pi's [28], [29]. However, none of them consider the overall memory constraints while distributing the NN, a fundamental limitation for severely resource constrained edge devices which prevents them from distributing larger NN benchmarks across edge nodes such as MSP430's (as discussed in Section VI).

Therefore, we propose DENNI an end-to-end automated approach that minimizes the number of distributed nodes while considering the total memory requirements including neural network weights, intermediate activations, and executable code. Our key insight in DENNI is to iteratively apply mixed integer non-linear programming (MINLP) to identify an intra- and cross-layer splitting (discussed further in Section III) that minimizes the number of nodes required for a NN's inference.

978-1-6654-4331-9/21/$31.00 ©2021 IEEE

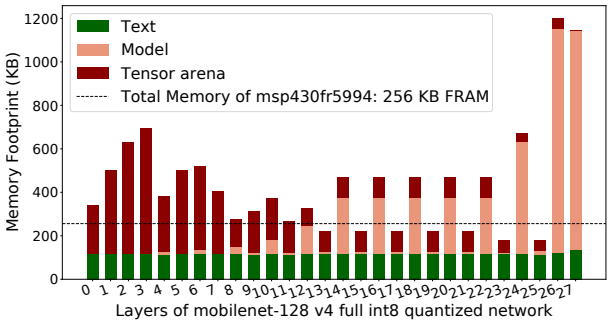


Fig. 1: MobileNet-128 v4 layer-wise memory footprint.

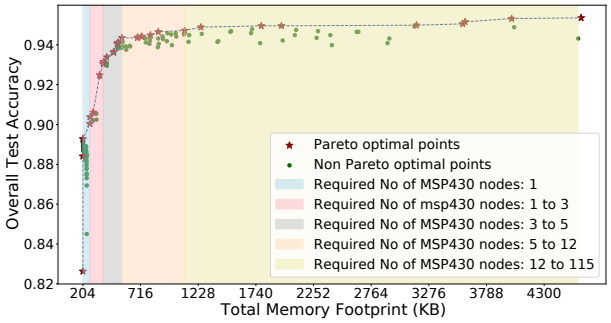


Fig. 2: HAR pruning: accuracy vs total memory footprint.

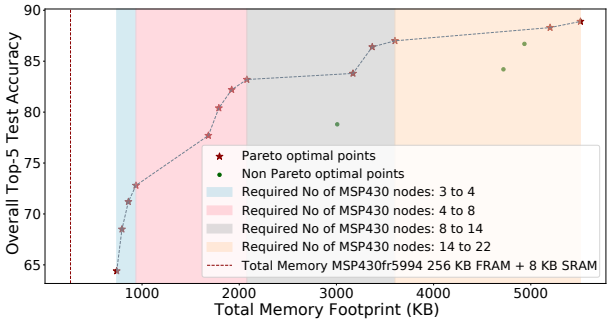


Fig. 3: MobileNet: accuracy vs total memory footprint.

Overall, we make the following contributions:

- We propose DENNI, an automated, end-to-end, holistic DNN inference splitting approach. DENNI utilizes MINLP to minimize the number of distributed nodes, supports all Tensorflow-lite micro layer operations, and statically distributes full, uncompressed, high-accuracy NN inference across severely resource-constrained edge nodes.
- We demonstrate DENNI is capable of enabling full-accuracy NNs for Human Activity Recognition, Keyword spotting, and ImageNet datasets on a network of MSP430 and CC2650-based nodes, which are too small to run these NNs on a single node.
- Our results show DENNI reduces total inference latency by 1.12× to 8.45× and worse-case node energy consumption by 1.73× to 59.21×.

Processors	NVM (KB)	SRAM (KB)	Freq (MHz)
ATSAML11E16A Rev. B	64	16	32
EFM32HG322F64 Rev. B	64	8	25
STM32L412 Rev. A	128	40	80
MSP432P401R Rev. C	256	16	48
MSP430FR5969	64	2	16
MSP430FR5994	256	8	16

TABLE I: Commercially available ultra-low power processors [30] suitable for IoT applications; Non-Volatile Memory: Flash/FRAM is in KB; frequency is in MHz.

II. Motivation

Edge devices are expected to lead the growth of individual intelligent devices during the IoT era. Processors in these devices are often physically co-located with sensors and have limited memory size and computational capabilities. Table I lists several commonly used edge processors [30]. These devices are severely resource constrained in both memory and compute capabilities. Thus, currently most data is either communicated directly to a datacenter in the cloud or to a nearby central gateway device where heavy analysis like NN inference is performed [17]. Subsequently, the results are communicated back to the IoT edge device for decision making.

However, offloading data to the datacenter or nearby gateway devices is expensive and suffers from unpredictable communication reliability that impacts overall performance and function. This is exacerbated by the growing number of edge devices. Moreover, sometimes (e.g., emergency disaster situations [31]) the central gateway device is not available. Accordingly, communicating sensed data directly to datacenter may fail, causing the IoT node to make incorrect decisions. Furthermore, clients increasingly want to keep data on the edge for security reasons [32]. Therefore, we focus on performing NN inference on local IoT edge devices.

Unfortunately, performing inference on edge devices is challenging due to their extremely limited memory and compute capabilities compared to the much higher resource requirements in many NN inference algorithms. Figure 1 breaks down the per-layer memory consumption for an 8-bit integer quantized version of MobileNet v4 [33], a popular mobile and edge NN [34]. Figure 1 shows almost all of its 28 layers are too big to fit on a single, resource constrained MSP430. Thus, the total memory footprint of popular NNs may be too large to fit on a single edge node. Interestingly, Figure 1 shows that the memory footprint of MobileNet's initial layers are mostly dominated by the size of the intermediate input-output tensors (i.e., data stored in the tensor arena), while

later layers are dominated by the memory of weights. While compression techniques such as pruning [35] and post training 8-bit quantization [36] can fit some NN inference models on a single node, the resulting accuracy is often significantly reduced. For example, Figure 2 compares the memory requirements for a set of compressed, pruned, and quantized NN designs to a 256 KB MSP430FR5994 for Human Activity Recognition (HAR) [37] on the UCI dataset [38]. The Pareto-optimal points are highlighted as stars. The best single-node design achieves 90% accuracy, 5% lower than the best overall accuracy (95.3%). However, using just three nodes increases accuracy to 93.1% (within 2.3% of the best). Figure 3 shows a similar trend for MobileNet.

The challenges in running HAR and MobileNet on edge devices with high accuracy highlights the need for a systematic approach to perform intra-layer splitting for layers with large weight memory footprints (e.g., layers 26-27 in Figure 1), cross-layer splitting for layers with large intermediate outputs (e.g., layers 0-10 in Figure 1), and both cross-layer and intra-layer successively for layers with large weight and intermediate output memory footprint (e.g., layer 14-15 in Figure 1). Further, given the high communication costs and unreliable inter-node communications, this approach must also minimize the number of nodes required. Accordingly, we propose DENNI, which applies MINLP to systematically optimize and distribute large NNs across edge devices.

III. Distributing NNs Across Edge Nodes

Figure 4 shows DENNI's toolflow. DENNI takes in an arbitrary NN as a Tensorflow-lite model, as well as a set of homogeneous target nodes to map the NN to. Based on the model's structure and the target node's constraints, DENNI's *distribution algorithm* identifies the model portion that should be mapped to each node to minimize the number of nodes required to perform inference using the model. The *model splitter* then applies these splits to the original NN, generating a new Tensorflow-lite model file containing the appropriate weights and biases for each node. Each model file is compiled along with the Tensorflow-lite micro source code to create a unique binary that is programmed onto a node.

A. Distribution Algorithm

The distribution algorithm takes an arbitrary NN such as Figure 5, which has N layers 0 to $N-1$, including K convolution, Q pooling and $N-K-Q$ fully connected layers,[1] and the memory constraints of a node, Mem_{node}. The distribution algorithm splits the NN across nodes such that the memory requirement of each split of the NN, Mem_{split}, is less than Mem_{node}. Moreover, the distribution algorithm tries to minimize the total number of nodes.

[1]DENNI supports the full range of layer operators specified by Tensorflow-lite micro, but we use this NN architecture for simplicity.

Algorithm 1: NN Distribution

Input: $\mathbf{L} = L_0, ..., L_{N-1}$, neural network define as list of N layers
Mem_{node}, the total memory size of the node
Output: $\mathbf{LC}$, list of layer chunks
for $i \leftarrow 0$ **to** $N-1$ **do**
  // get the number of nodes required for this layer and associated parameters;
  $(NL_i, P_i) \leftarrow layerMINLP(L_i, Mem_{node})$;
end
empty list of layer chunks, $\mathbf{LC}$;
previous min nodes, $PrevNLC \leftarrow NL_0$;
previous parameters, $PrevP \leftarrow P_0$;
current layer chunk first node, $j \leftarrow 0$;
for $k \leftarrow 1$ **to** $N-1$ **do**
  $(CurrNLC, CurrP) \leftarrow chunkMINLP(\mathbf{L_{j:k}}, Mem_{node})$;
  if $CurrNLC < PrevNLC + NL_k$ **then**
    $PrevNLC \leftarrow CurrNLC$;
    $PrevP \leftarrow CurrP$;
  else
    $\mathbf{LC}.add((PrevNLC, CurrP))$;
    $PrevNLC \leftarrow NL_k$;
    $PrevP \leftarrow P_k$;
    $j \leftarrow k$;
  end
  $\mathbf{LC}.add((PrevNLC, CurrP))$;
end

Algorithm 1 shows the distribution algorithm. First, our distribution algorithm performs intra-layer splitting, finding the minimum number of nodes per layer using a Mixed Integer Non-Linear Programming (MINLP) formulation [39]. In order to further reduce the number of nodes, our distribution algorithm greedily considers cross-layer and intra-layer co-optimization for a group of adjacent layers, called *layer chunks*, starting from the input layer. We use a greedy approach since trying all possible layer combinations to co-locate would be intractable as the number of layers increases. Only considering adjacent layers provides good solutions because the adjacent layers often share input/output memory values and it allows the optimizer to tradeoff duplication of weights within a layer with reduced activation memory. Again, we use MINLP to solve for the minimum number of nodes needed by layer chunk $L_{j:k}$. The distribution algorithm creates successively larger layer chunks until the minimum number of nodes for the layer chunk, $CurrNLC$, is larger than the sum of the required nodes for the previous layer chunk, $PrevNLC$, and the layer-

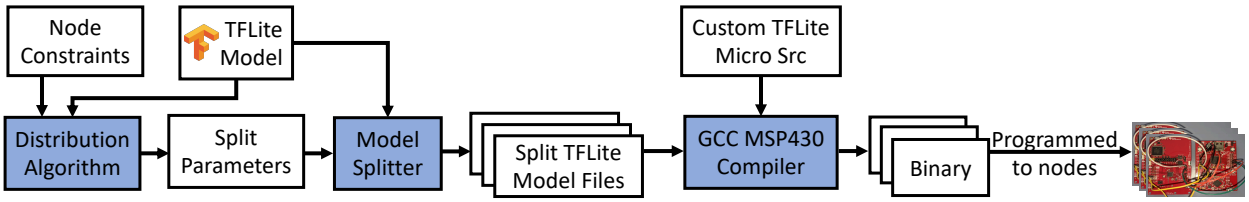


Fig. 4: DENNI toolflow.

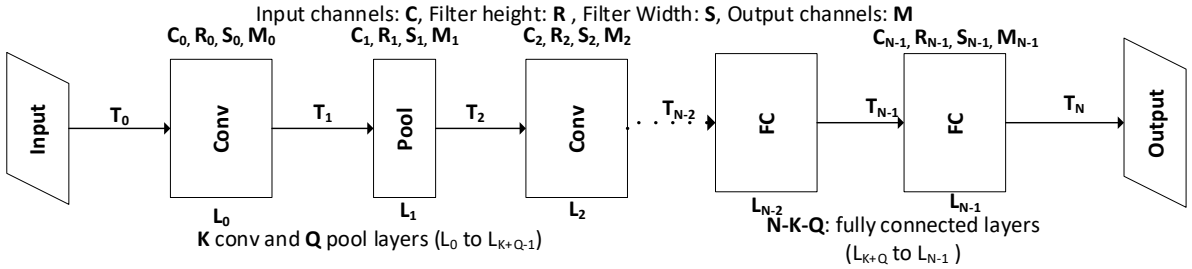


Fig. 5: A representative neural network.

wise optimal solution of the next layer, NL_k. Once this happens, we save the distribution parameter assignment for that layer chunk, $CurrP$.

1) Memory Requirements: The MINLP optimization must calculate the memory requirements for arbitrary splits of a NN. A Tensorflow-lite micro NN has three memory components – model parameters, tensor arena, and the actual program text segment (i.e., "code").

The model parameters include the weights and biases for each layer. The size (in bytes) of these parameters, W_i, for any convolutional or fully connected layer i in an `int8`-quantized NN (with 32-bit integer biases) is:

$$W_i = C_i * R_i * S_i * M_i + 4 * M_i$$

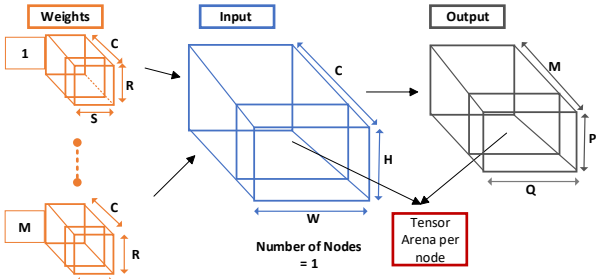
where C_i, R_i, S_i, and M_i denote the number of input channels, filter height, filter width and output channels, respectively, for the layer i of the NN (see Figure 5).[2]

The tensor arena holds activations for each layer (e.g., input, output, and intermediate feature maps). We denote these intermediate tensors as T_i. To determine the tensor arena size, we must consider both the sum of the largest pair of adjacent activation tensors, (T_i, T_{i+1}), and the quantization parameters (a 32-bit integer zero point and a 32-bit float scale for each channel). Thus, the worst-case tensor arena may be calculated as:

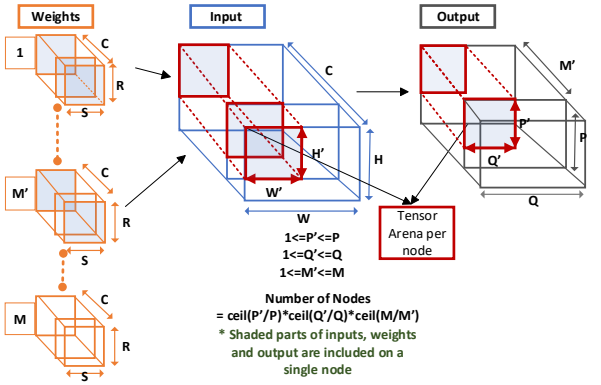
$$\text{tensor arena} = \max_{i \in [0, N-2]} (T'_i + T'_{i+1}) + 2*4*(\sum_{i=0}^{N} M_i + 1)$$

Here for a particular NN the code size is constant, and depends upon the various NN inference operators.

[2]For depthwise convolution, $M_i = 1$ and biases will be C_i. For a fully connected layer, we have $R_i = H_i$ and $S_i = W_i$.



(a) Unsplit convolution layer.



(b) Intra-layer split reducing tensor arena and weight size.

Fig. 6: Intra-layer MINLP optimization.

Therefore, the total memory footprint M_N for NN inference becomes:

$$Mem_{inf} = \sum_{i=0}^{N-1} W_i + tensor\,arena + code \quad (1)$$

2) Intra-Layer MINLP Optimization: Given Equation 1, to reduce a layer's memory consumption we can: (1) reduce model size and/or (2) reduce tensor arena size.

Within a layer, such as the one shown in Figure 6a, we allow splitting the weights by output channel, M, input channel, C, or within a filter, R or S. This results in an intra-layer splitting as shown in Figure 6b.[3]

We only allow splitting for the output channel since the other forms of splitting require more communication prior with the next layer (often a non-linear layer such as ReLU). Splitting by the output channel reduces the weights memory by $ceil(M/M')$ where M' is the splitting parameter selected by the optimizer.

Within a layer we split the activation tensors (input and output activations) by the output channel, M, or within a channel, H or W. We do not allow splitting by input channel for the same reason as above. Thus, the tensor arena area for a node in this layer is reduced to:

$$\text{tensor arena} = H' * W' * C + P' * Q' * M' \quad (2)$$

where $1 <= P' <= P$ and $1 <= Q' <= Q$ dictate the value of H', W'.

Given the above parameters, for a given layer the number of nodes required for splitting is:

$$NL_l = ceil(P/P') * ceil(Q/Q') * ceil(M/M') \quad (3)$$

3) Cross- and Intra-Layer MINLP Co-Optimization: As shown in Figure 3, the tensor arena dominates the memory usage of many early layers of NNs. For such layers, optimizing across layers can reduce the tensor arena size since the memory used by the input activations of one layer can be reused by the output activations of the next layer(s). For example, in the NN layers in Figure 7a, part of T_0's area can be reused by part of T_2 so long as those parts are both executed on the same node. Intra-layer splitting must also be considered in order to split the tensor arenas across nodes while balancing the increasing number of weights.

Figure 7b depicts an example of cross- and intra-layer co-optimization of a layer chunk consisting of two layers: a depthwise convolution followed by a convolution. Starting with the last layer, L_1, we parameterize splitting of the output tensor arena (P'_1, Q'_1) where $1 <= P'_1 <= P_1$ and $1 <= Q'_1 <= Q_1$. Similarly, depending upon the filter's R_i and S_i values, for each layer we backtrack until the initial layer of the layer chunk. Since the tensor-arena depends upon the maximum pair of the intermediate input-output tensor, as given by (P'_1, Q'_1), the overall arena may be reduced. However, this requires replicating the weights across $ceil(P_1/P'_1) * ceil(Q_1/Q'_1)$ nodes. Note that only P'_1 and Q'_1 are free parameters since P'_0 and Q'_0 must be sized such that all the necessary input activations are present on the node to compute P'_1 and Q'_1. Since these values will grow larger they end up limiting the reduction of tensor arena size per node as the layer chunk increases in depth.

Although we do not allow input channel splitting for convolution layers, we do allow it for depthwise convolution layers since they have relatively large weights compared with convolutional layers and thus would limit the benefit of increasing layer chunk depth. Figure 7b demonstrates this: the C'_0 and M'_1 output channels from C_0, M_1 are mapped to each node. However, this requires some inter-node communication within a layer chunk (shown by the green volume in tensor T_1^{**} in Figure 7b) since in the original network $C_0 = C_1$. This additional communication does not significantly impact the latency (see Figure 10) while reducing the number of nodes required. For such cases, the tensor arena is the large of the two: T_1^{**}. Thus, the total tensor arena for layer-chunk can be calculated by:

$$\text{tensor arena} = \max_{i \in [0,2]} (T'_i + T'_{i+1})) \quad (4)$$

Given the above splitting parameters for the two layers, the number of nodes required to support the splitting is given by:

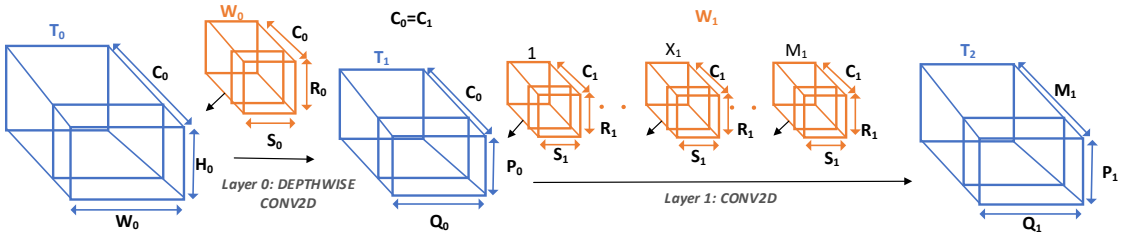
$$NLC_{0,1} = ceil(P_1/P'_1) * ceil(Q_1/Q'_1) * Max(ceil(C_0/C'_0), ceil(M_1/M'_1)) \quad (5)$$

We generalize this approach to an arbitrary number of layers within a layer chunk as required by our distribution algorithm.

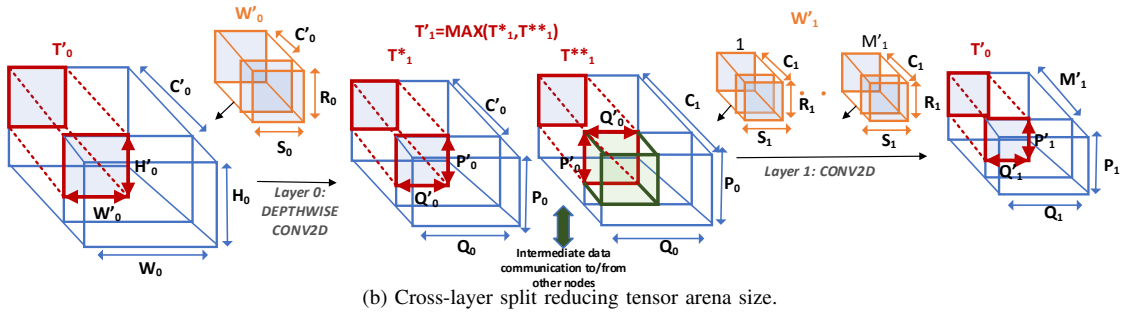
IV. Methodology

We use MSP430FR5994-based nodes with 256KB of memory [40] as our edge node. This node is popular in energy-harvesting systems due to its on-chip FRAM non-volatile memory that allows 1-2 cycle reads and writes without having to erase a sector. Each node is programmed with a unique bare-metal binary produced by DENNI. These binaries contain a custom 16-bit implementation of Tensorflow-lite micro along with the appropriately partitioned model parameters. We measure the actual runtime computation cost on MSP430FR5994s executing the partitioned NN inference binaries using the energy trace utility [41]. We analyze the inference of six modern NNs, as highlighted in Table II.

[3]We show a commonly used convolutional layer. Fully-connected layers are treated as convolutional layers with $H = S$ and $W = R$. Since pooling and non-linear activation layers are applied element-wise, we aggregate them into the previous layer. However, they have little impact since they do not require additional weights or tensor arena space. Finally, we output split layers of operators such as softmax, but perform softmax on a single node for mathematical correctness.



(a) Example layer chunk with two layers – a depthwise convolution layer followed by a convolution layer.



(b) Cross-layer split reducing tensor arena size.

Fig. 7: Cross- and intra-layer co-optimization.

Dataset	ImageNet [34]				UCI [38]	DSCNN [42]
NN	MNv1	MNv2	MNv3	MNv4	HAR	DSCNN
Layers	28	28	28	28	3	12
Resolution	128	128	128	128	128*6	49*10
Width multiplier	0.25	0.5	0.75	1	–	–
Million MACs	14	49	104	186	–	–
Tensor arena (KB)	154	249	345	443	57	136
Model size (KB)	486	1333	2563	4177	6311	518
Code size (KB)	145	145	145	145	141	151
Top-5 Accuracy	64.4%	77.7%	78.8%	85.8%	95.3% (Top1)	94.58% (Top1)

TABLE II: Benchmarks used.

To enable BLE communication between our edge nodes, we use the CC2640R2F [43]. We used SPI communication to communicate between the MSP430 and the CC2640R2F. Figure 4 shows our node test setup. Since we experiment with numerous partitioning schemes involving up to hundreds of nodes, we modeled communication latency rather than manually setting up the topology for each experiment. We assume all nodes are connected in a fully-connected topology. Based on measurements of our system, it takes 11.6 microseconds per byte of data, which corresponds to 84.186 KB/s, a reasonable rate for BLE 5 supported by CC2640R2F. Moreover, based on prior work we model our communication energy as 1 Joule per 350 KB for a maximum Protocol Data Unit (PDU) of 65 bytes [44].

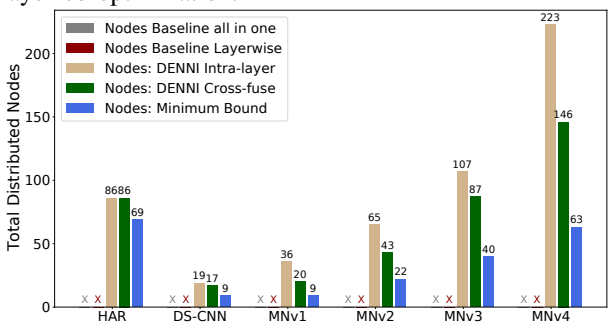


Fig. 8: Total number of nodes required by each NN distribution approach. DENNI supports distribution of NN's across severely resource-constrained nodes with only 256KB memory available; 'X' means the benchmark cannot run using that approach.

V. Results

A. Minimum Number of Nodes

To ensure that DENNI finds a valid distribution for all benchmarks, we compare DENNI to the minimum number of nodes within available solutions. Figure 8 shows the number of nodes each approach requires to provide NN inference at full accuracy. *Baseline* represents the case where only a single node is used for a NN. However, none of our benchmarks can fit on a single node at full accuracy, represented with an 'X'. We also compare DENNI against *Layerwise Splitting*, which distributes an NN layerwise across nodes, as in prior work [22], [45]. However, since each NN has at least one layer that is larger than a single node memory, *Layerwise Splitting* fails. DENNI utilizes *Intra-Layer Splitting* to overcome

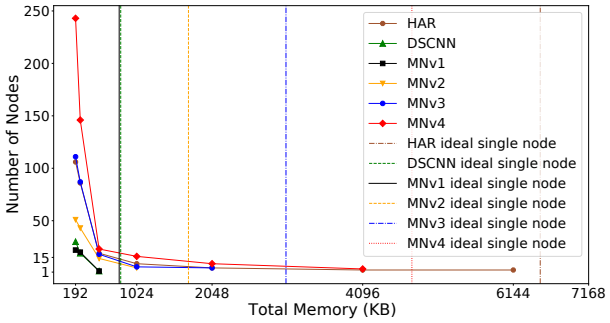


Fig. 9: DENNI MINLP number of nodes vs node memory scaling across benchmarks.

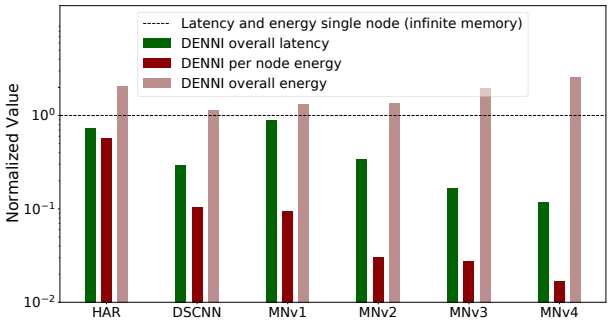


Fig. 10: Inference latency and energy using DENNI on MSP430FR5994(256 KB) normalized to values of a single node with infinitely large memory.

this and successfully, automatically distributes each NN with intra-layer splitting, requiring 36 to 223 nodes.[4] By using cross-layer optimizations in conjunction with intra-layer splitting, *DENNI Intra- and Cross-Layer*, reduces the required number of nodes by up to 35%.

Minimum Bound represents a fundamentally unachievable lower bound on the number of nodes. We calculate it by dividing the sum of a neural network's total baseline model size and baseline tensor arena by the data memory capacity of a single node. DENNI's intra- and cross-layer approach provides the closest solution to the minimum bound in all cases. However, the minimum bound is not achieved because of splitting overheads – duplicate weights to support cross-layer distribution, duplicate tensor arena area to store intermediate results on each node, additional code for pad operator and splitting, and unused space on a node (i.e., a form of memory fragmentation caused by discrete layer sizes not perfectly matching node memory size).

Other work maps NNs to resource-constrained nodes using pruning, compression, or different architectures [6]–[16]. However, in order to fit on a single node, these approaches often come with significant accuracy reductions (see Section II). DENNI is an orthogonal approach that can be applied on top of such optimizations in order to perform inference in a sensor network at a desired accuracy. For example, we show several different versions of MobileNet that have varying accuracy and memory requirements. Although they are all too large for a single node, DENNI can successfully perform inference by distributing them across multiple nodes.

B. Node Memory Scalability

In the future, as technology used in such resource-constrained nodes evolves (i.e., as density increases from scaling or new memory technology), node memory will likely continue to increase, yet DENNI can still provide benefits by cross fusing more layers in layer chunks and hence saving arena size and exploiting larger parallelism due to big layer chunk sizes compared to other manual approaches. Figure 9 shows DENNI can be successfully applied to increasing node memory sizes until the min-nodes estimated by DENNI approaches a single node and node memory becomes large enough for each benchmark to fit on one node (up to 6MB).

C. Latency and Energy

Figure 10 shows the overall inference latency and the per-node inference energy. Overall, distributing the NNs across nodes reduces inference latency by 1.12× to 8.45× relative to a single node MSP430 with arbitrarily large memory.[5] DENNI's inference latency ranges from to 2.7 and 15.8 minutes overall inference time. Although distributing a NN across multiple nodes increases communication costs (e.g., more communication within a layer chunk) and also have some re-computation overhead (e.g. recomputing adjacent overlapping feature maps across different input split partitions of same layer chunk), DENNI's distributed approach increases parallelism (e.g., nodes within the same layer chunk) and reduces overall inference latency. This occurs due to the relative density of computation within layer chunks compared with communication between layer chunks and redundant computation overhead.

The inference energy varies across each node since the amount of computation and communication vary. Figure 10 shows the average per node energy relative to a single node with arbitrarily large memory. In all cases, DENNI reduces per node energy from 1.73× to 59.21×. The worst-case node energy is 0.98 Joules, which corresponds to a 1.73× decrease compared to the

[4]The benchmarks produce bitwise identical results when run on MSP430 nodes compared with the original, unsplit Tensorflow-lite micro model's execution on a host computer.

[5]We conservatively estimate such a hypothetical node by measuring the latency of all the required computational components and then summing them together.

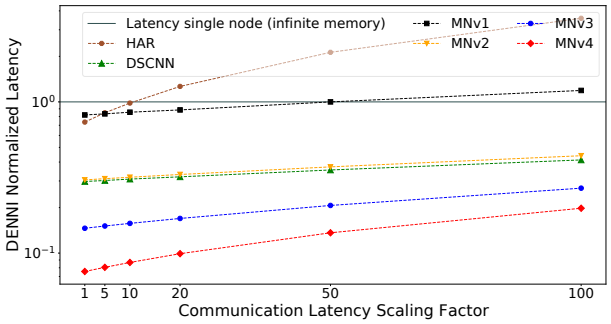


Fig. 11: DENNI overall latency normalized to a single node with infinitely large memory.

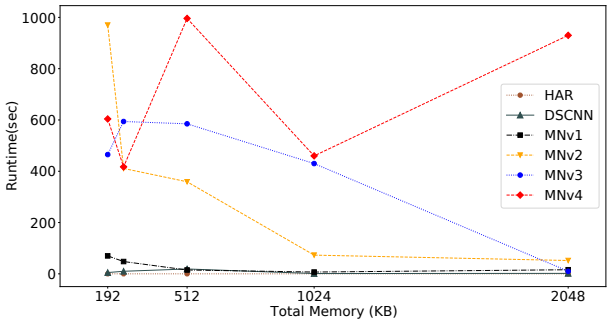


Fig. 12: MINLP solver total runtime (X86_64 PC) vs node memory across benchmarks.

energy cost if run on a single arbitrarily large memory node. Overall, largely due to communication energy and redundant calculations across nodes, the total energy required is increased by 158%. However although the overall inference energy increases, this energy cost is distributed across individual nodes and is relatively small value per node. This trade-off is amenable to energy-harvesting sensor networks based on batteryless devices [46]–[49] where spare harvested energy of remote nodes may be harnessed for distributed inference when cloud connectivity may not be present.

D. Communication Cost

When inference is distributed, we observe that most of the overall inference energy increases is due to communication energy. One of the many possible ways we may be able to significantly reduce the energy cost of inference is by using techniques such as backscatter [50] for communication rather than the active radio communication used in Section IV. By reflecting a carrier radio signal from a transmitter, backscatter could be 1000× more efficient than traditional active radio communications such as BLE [50]. However, such energy efficiency increases need reduced transmission bit rate, resulting in increased communication latency. Nevertheless, even in that scenario DENNI still offers considerable overall inference latency benefits. Figure 11 shows how DENNI performs when we increase the modeled cost of communication latency by up to 100× its baseline value while keeping the actual measured computation latency cost unchanged. Barring less compute intensive benchmarks such as MNv1 and HAR, DENNI provides a consistent speed up of more than 3× compared to the case when the inference is executed on a MSP430 with arbitrarily large memory for compute intensive benchmarks such as DSCNN, MNv2, MNv3, and MNv4.

Overall, DENNI is effective at distributing arbitrary NNs across extremely resource-limited edge nodes. The distribution results in lower overall latency and per-node energy even as node memory size or communication latency costs increase for energy-efficiency. As a result, DENNI can enable pervasive intelligence scenarios when the cloud/gateway connectivity may not be present at all times such as disaster relief or low orbit constellation of energy-harvesting nano-satellites in space [51].

E. MINLP Cost Analysis

The MINLP solver time can significantly increase as more constraints are added. Instead of solving for the minimum nodes for an entire NN, which could produce an intractable MINLP problem, DENNI uses an iterative approach starting from the input layers as described in Section III). As shown in Figure 12, this results in a scalable solution that completes in tens of minutes across all benchmarks. MINLP is implemented using the Gekko optimization suite [39] and the APOPT solver [52]. For execution we use an Intel i7 x86_64 6-core, 2.2 GHz CPU having 32 GB's of RAM. Unlike some other edge-only approaches described in Section VI, MINLP does not require offline layerwise profiling of the NN nor does it require online monitoring and analysis — DENNI's MINLP overhead is one-time and offline.

VI. Related Work

Prior work examines fitting inference on a single node by scaling the model, offloading part of the work to relatively resource-rich devices (e.g., cloud or edge servers), and distributing NNs across multiple nodes.

Single-node Model Scaling: To fit ML models into a single node, prior approaches use a combination of neural architecture search (NAS) [6], [8], compression via pruning and quantization [9], [35], [53], [54], NN-specific approaches [10], [13]–[16], and non-NN approaches that use techniques like low dimension projection [11], [12]. However, unlike DENNI such techniques decrease NN inference accuracy, require costly retraining (often with custom NN-specific implementations), and may still need more memory than available on severely resource constrained devices.

Single-node Offloading: An alternative approach uses model partitioning to divide the model between edge device and powerful cloud systems such that a part of computation is offloaded to cloud [20], [55]–[57]. A twist on this approach, early-exit models [19], [58]–[60], allows the resource-constrained device to opportunistically skip some layers, lessening the compute and memory requirements for certain inputs which can be classified easily with a high degree of confidence. For other inputs that need the full NN, a cloud device would need to be used. As proposed, offloading approaches require communication of data out of the network of resource-constrained devices. Additionally, the desired portion of the NN performed on the local node may still not fit in the memory of a resource-constrained node.

Multi-node Edge-only: Table III compares DENNI to other, prior approaches that split NNs across multiple edge nodes, across several important features: support for splitting convolution layer by arenas, weights, and both arenas & weights, distributing FC layers by weights, and which benchmarks in Table II they support. MoDNN distributes arena-dominated convolution layers using input and output splitting for FC layers [22]. Similarly, collaborative inference utilizes input, output, and spatial splitting for convolution and FC layers [45]. However, these approaches distribute each layer to unique nodes which drastically increases the number of nodes and communication costs. Compared to collaborative inference, DeepThings, Edge Intelligence, and Fully Dist. [26]–[28] save layerwise synchronization overhead by cross fusing the initial convolutional layers. However, none of them consider the memory constraint of edge nodes while minimizing, limiting their use in deployments on severely resource constrained edge devices. Finally, DeepSlice partitions convolutional NNs using memory reclamation to reduce footprint [29]. However, despite having a similar greedy memory allocation policy that only uses maximum memory of the sum of adjacent tensors (e.g., the tensor arena in Tensorflow-lite in Section III), the resulting NN inference partitions would still not fit on severely resource constrained edge nodes without taking into consideration node memory constraint while distributing and they are still not able to partition layers with both high arena and weight memory.

As a result, none of the existing approaches are able to enable complete inference for all the benchmarks as shown in Table III on severely resource constrained edge devices in Table I. Unlike the prior work, DENNI supports all of these benchmarks and features by utilizing a lightweight MINLP approach that considers memory constraints while also providing a scheme to perform both arena and weight splitting for all layer types.

Related work	**Conv-arena**	**Conv-weights**	**FC-weights**	**Conv-arena & weights**	**Benchmarks Table II**
MoDNN [22]	✓	✗	✓	✗	None
Deep Things [28]	✓	✗	✗	✗	None
Collab. inf. [45]	✓	✓	✓	✗	HAR, MNv1
Fully dist. [27]	✓	✓	✓	✗	HAR, MNv1
Deep Slice [29]	✓	✓	✓	✗	HAR, MNv1
DENNI	✓	✓	✓	✓	All

TABLE III: Comparison to prior NN splitting work.

VII. Conclusion and Future Work

The growing memory requirements of NNs are at odds with mapping to extremely resource-constrained nodes such as those suitable for energy-harvesting sensor networks. Even a single layer within a pruned NN may exceed the memory capacity of a single node. Therefore, we propose DENNI, which combines MINLP with intra- and cross-layer distribution approaches to enable NN inference on such resource-constraint edge devices.

DENNI provides a platform to investigate open research questions for inference at the extreme edge. For example, DENNI can explore mechanisms that provide dependable inference on energy-harvesting nodes which may be *intermittently* powered due to the inherent variability of energy-harvesting mechanisms. DENNI also provides a potential base for lightweight dynamically adaptative NN splitting. In addition, we envision extending DENNI to additional machine learning workloads such as RNNs and online learning in order to expand the capabilities of learning at the extreme edge.

Acknowledgment

This work was supported in part by the U.S. National Science Foundation under Grant 2008548.

References

[1] D. Schatsky *et al.*, "Pervasive intelligence Smart Machines Everywhere," 2019.
[2] Y. Zhang *et al.*, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.
[3] Mckinsey, "Growing opportunities in the Internet of Things," 2021.
[4] ABI Research, "Hardware Vendors Will Win Big in Meeting the Demand For Edge AI Hardware," 2018.
[5] J. Cheng *et al.*, "Recent advances in efficient computation of deep convolutional neural networks," *FITEE*, 2018.
[6] C. Banbury *et al.*, "MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers," in *MLSys*, 2021.
[7] Y. Cherapanamjeri *et al.*, "Thresholding based Efficient Outlier Robust PCA," 2017.

[8] I. Fedorov *et al.*, "SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers," in *NeurIPS*, 2019.

[9] I. Fedorov, M. Stamenovic *et al.*, "Tinylstms: Efficient neural speech enhancement for hearing aids," *arXiv preprint arXiv:2005.11138*, 2020.

[10] S. Gopinath *et al.*, "Compiling KB-Sized Machine Learning Models to Tiny IoT Devices," in *PLDI*, 2019.

[11] C. Gupta *et al.*, "ProtoNN: Compressed and Accurate KNN for Resource-Scarce Devices," in *ICML'17*, 2017.

[12] A. Kumar *et al.*, "Resource-Efficient Machine Learning in 2 KB RAM for the Internet of Things," in *ICML*, 2017.

[13] A. Kusupati *et al.*, "FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network," in *NeurIPS*, 2018.

[14] S. G. Patil *et al.*, "GesturePod: Enabling On-Device Gesture-Based Interaction for White Cane Users," in *UIST*, 2019.

[15] T. Tambe *et al.*, "9.8 A 25mm2 SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET," in *ISSCC*, 2021.

[16] U. Thakker *et al.*, "Doping: A technique for Extreme Compression of LSTM Models using Sparse Structured Additive Matrices," in *SysML*, 2021.

[17] J. McChesney *et al.*, "DeFog: Fog Computing Benchmarks," in *SEC*, 2019.

[18] T. Bolukbasi *et al.*, "Adaptive neural networks for efficient inference," in *ICML*, 2017.

[19] S. Teerapittayanon *et al.*, "Distributed deep neural networks over the cloud, the edge and end devices," in *ICDCS*, 2017.

[20] J. H. Ko *et al.*, "Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms," in *AVSS*, 2018.

[21] E. Li *et al.*, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *MECOMM*, 2018.

[22] J. Mao *et al.*, "Modnn: Local distributed mobile computing system for deep neural network," in *DATE*, 2017.

[23] J. Mao, Z. Yang *et al.*, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *ICCAD*, 2017.

[24] Z. Zhou *et al.*, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing," *arXiv preprint arXiv:1905.10083*, 2019.

[25] N. D. Lane *et al.*, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *IPSN*, 2016.

[26] L. Zhou *et al.*, "Distributing deep neural networks with containerized partitions at the edge," in *HotEdge*, 2019.

[27] R. Stahl *et al.*, "Fully Distributed Deep Learning Inference on Resource-Constrained Edge Devices," in *SAMOS*, 2019.

[28] Z. Zhao *et al.*, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *TCADICS*, 2018.

[29] S. Zhang *et al.*, "Deepslicing: Collaborative and adaptive cnn inference with low latency," *TPDS*, vol. 32, no. 9, 2021.

[30] Embedded Microprocessor Benchmark Consortium. Embedded microprocessor benchmark consortium. [Online]. Available: https://www.eembc.org/ulpmark/scores.php

[31] B. K. Dar *et al.*, "An Architecture for Fog Computing Enabled Emergency Response and Disaster Management System (ERDMS)," in *ICAC*, 2018.

[32] Z. Whittaker, "After Spate of Recent Attacks, Google beefs up Nest security protections," 2020.

[33] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[34] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," *Advances in NIPS*, vol. 25, 2012.

[35] H. Hu *et al.*, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[36] Google. Post-training integer quantization. [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_integer_quant

[37] A. Ignatov, "Real-time Human Activity Recognition from Accelerometer Data using Convolutional Neural Networks," *Applied Soft Computing*, vol. 62, 2018.

[38] U. M. L. Repository. Human Activity Recognition Using Smartphones Data Set. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

[39] L. D. Beal *et al.*, "Gekko optimization suite," *Processes*, 2018.

[40] *MSP430FR5994 LaunchPad™ Development Kit (MSP-EXP430FR5994)*, Texas Instruments.

[41] Texas Instruments. EnergyTrace Technology. [Online]. Available: http://www.ti.com/tool/ENERGYTRACE

[42] ARM. ARM software ML zoo. [Online]. Available: https://github.com/ARM-software/ML-zoo/tree/master/models/keyword_spotting/ds_cnn_large/tflite_int8

[43] *CC2640R2F SimpleLink™ Bluetooth 5.1 Low Energy Wireless MCU*, Texas Instruments.

[44] M. Siekkinen *et al.*, "How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4," in *WCNCW*, 2012.

[45] R. Hadidi *et al.*, "Toward collaborative inferencing of deep neural networks on internet-of-things devices," *IOT-J*, vol. 7, no. 6, 2020.

[46] V. Deep *et al.*, "Harc: A heterogeneous array of redundant persistent clocks for batteryless, intermittently-powered systems," in *RTSS*, 2020.

[47] M. Wymore *et al.*, "Lifecycle management protocolsfor batteryless, intermittent sensor nodes," in *IPCCC*, 2020.

[48] V. Kortbeek *et al.*, "Time-sensitive intermittent computing meets legacy software," in *ASPLOS*, 2020.

[49] K. Maeng *et al.*, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *PLDI*, 2020.

[50] Xu *et al.*, "Practical backscatter communication systems for battery-free internet of things: A tutorial and survey of recent research," *IEEE Signal Processing Magazine*, vol. 35, no. 5, 2018.

[51] B. Denby *et al.*, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *ASPLOS*, 2020.

[52] J. Hedengren *et al.*, "Apopt: Minlp solver for differential and algebraic systems with benchmark testing," in *Proceedings of the INFORMS National Meeting*, vol. 1417, 2012.

[53] S. Han *et al.*, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv preprint arXiv:1510.00149*, 2015.

[54] Y. He *et al.*, "Channel pruning for accelerating very deep neural networks," in *ICCV*, 2017.

[55] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, Apr. 2017.

[56] H.-J. Jeong *et al.*, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *SoCC*, 2018.

[57] D. Hu *et al.*, "Fast and accurate streaming cnn inference via communication compression on the edge," in *IoTDI*, 2020.

[58] S. Scardapane *et al.*, "Why should we add early exits to neural networks?" *Cognitive Computation*, vol. 12, no. 5, 2020.

[59] N. Passalis *et al.*, "Efficient adaptive inference for deep convolutional neural networks using hierarchical early exits," *Pattern Recognition*, vol. 105, 2020.

[60] Li *et al.*, "Deep learning for smart industry: Efficient manufacture inspection system with fog computing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, 2018.