2023 IEEE International Conference on Edge Computing and Communications (EDGE) | 979-8-3503-0483-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/EDGE60047.2023.00029

Creating Robust Deep Neural Networks With Coded Distributed Computing for IoT

Ramyad Hadidi[§]
Rain AI
ramyad@rain.ai

Jiashen Cao
Georgia Tech
jiashenc@gatech.edu

Bahar Asgari[§]
University of Maryland
bahar@umd.edu

Hyesoon Kim
Georgia Tech
hyesoon.kim@gatech.edu

***Abstract*—The increasing interest in serverless computation and ubiquitous wireless networks has led to numerous connected devices in our surroundings. Such IoT devices have access to an abundance of raw data, but their inadequate resources in computing limit their capabilities. With the emergence of deep neural networks (DNNs), the demand for the computing power of IoT devices is increasing. To overcome inadequate resources, several studies have proposed distribution methods for IoT devices that harvest the aggregated computing power of idle IoT devices in an environment. However, since such a distributed system strongly relies on each device, unstable latency, and intermittent failures, the common characteristics of IoT devices and wireless networks, cause high recovery overheads. To reduce this overhead, we propose a novel robustness method with a close-to-zero recovery latency for DNN computations. Our solution never loses a request or spends time recovering from a failure. To do so, first, we analyze how matrix computations in DNNs are affected by distribution. Then, we introduce a novel coded distributed computing (CDC) method, the cost of which, unlike that of modular redundancies, is constant when the number of devices increases. Our method is applied at the library level, without requiring extensive changes to the program, while still ensuring a balanced work assignment during distribution.**

***Index Terms*—Edge AI, Reliability, IoT, Edge, Distributed Computing, Collaborative Edge & Robotics**

I. Introduction

Recent years have witnessed the emergence of deep neural network (DNN) applications. Additionally, with the proliferation of Internet-of-Things (IoT) devices, they became inseparable from our daily lives. The conventional methods to process raw IoT data are to offload them to cloud services. However, moving such a tremendous amount of data incurs a high amount of monetary cost and delay, besides creating a major concern of privacy leakages. Therefore, serverless and edge computation paradigms are recognized as promising solutions. As a result, pushing the frontier of DNNs computations to the edge is receiving a tremendous amount of interest both from academia [1]–[9] and from the industry with commercial edge-tailored hardware accelerators such as NVIDIA Jetson Nano, edge TPU, and Intel Movidius.

Processing IoT data locally in the edge may suffer from poor performance and energy efficiency because the computational demand from DNNs outweighs the computation capacity and energy constraints of IoT devices. Furthermore, the computational demands are escalated because these devices have to meet real-time constraints. Even for edge-tailored hardware accelerators, the real timeliness of applications is not guaranteed [10], [11]. Nevertheless, privacy concerns, unreliable connection to the cloud, tight real-time requirements, and personalization are still pushing inferencing to the edge. To address the resource constraint challenges, a solution is to distribute heavy computations among idle devices [1], [2], [4], [12] because the state-of-the-art IoT networks are formed with various IoT sensors and recording agents, such as HD cameras and temperature sensors, many of which are capable of performing computations. However, such a distribution is susceptible to failures, from short disconnectivity and user interaction to losing a device. This fact necessitates developing a robust method for tolerating these failures. Additionally, since IoT networks use wireless technology, unreliability and variability in their networks are much higher than acceptable limits to ensure a robust system.

We extend studies that enable distributed single-batch inference of DNNs in the edge [1], [2], [4], [12] to tolerate failures with close-to-zero recovery latency. We first analyze general methods of distributing the computations of DNNs and how their underlying general matrix-matrix multiplication (GEMM) is affected by distribution. Such a detailed study is necessary to introduce a general seamless method within the underlying library or machine learning framework. Then, we propose a new recovery method based on coded distributed computing (CDC) that enables distributed DNN models on IoT devices to tolerate failures. Our method is inspired by CDC applications in big data analytics [13], and speeding up distributed learning using codes [14].

To enable robustness in distributed IoT, we introduce an extra coded computation per device. We propose a novel fault recovery method based on CDC that has close-to-zero recovery latency, does not disturb the balanced work assignment in distribution, requires minimal changes to the program, and has a constant cost with the increasing number of devices. Our introduced extra computations are derived by thoroughly analyzing how general methods of distributing the computation of DNNs affect their underlying GEMM. The added computations are similar in nature to those of DNNs, which eases balancing the work among IoT devices and reduces the deployment cost. Balanced distribution is essential in attaining the expected performance. Additionally, since our method is implemented at the library level, it does not require changes to the program. Moreover, unlike approaches that sacrifice latency for robustness to recompute the missing part of the data, our

This work was supported in part by the NSF grant number 2103951.
[§] This work was done when the authors were affiliated with Georgia Tech.

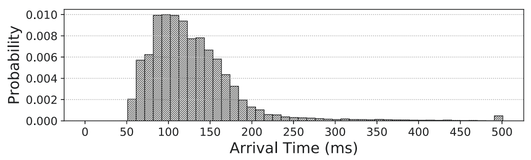


Fig. 1: Arrival time histogram of data packets in a WiFi network for a four-device IoT system with RPis.

method, even at the time of failures, provides close-to-zero recovery time, which is necessary for critical time-sensitive tasks. Finally, compared with conventional modular redundancy methods with redundant computation by introducing a linear number of additional devices, our method has a constant cost as the number of devices increases. We demonstrate our method on Raspberry Pis (RPis), which represents the de facto choice for several small and edge use cases.

II. Motivation

To illustrate unreliability in the communication latency of IoT systems, Figure 1 shows a histogram of the arrival times for data packets in an IoT system of four RPis (system setup in Section IV). This system performs the computation for a fully-connected (FC) layer of size 2048 in a distributed fashion and waits for the response, the measured time for the computation on a single device is 50 ms. This is why, in Figure 1, no packet arrives earlier than 50 ms. As seen, around 34% of the arrival times is within 100 ms, and 42% is within 150 ms. Even after 2x the computation time, around 34% of the packets have not arrived yet. Such behavior in distributed systems causes *straggler problem*, in which the slowest node in the distributed system defines the total latency. Our method, by introducing robustness in such systems, can additionally alleviate the straggler problem while also guaranteeing close-to-zero recovery latency.

To understand how failures are destructive in DNN applications, we perform another set of experiments, in which some part of data within a layer is lost. We choose two models: LeNet-5 [15] and Inception v3 [16]. LeNet-5 is a simple model that detects handwritten digits from 10 classes and consists of only five layers. On the other hand, Inception v3 is a DNN model for image recognition for 1k classes with 159 layers. Figure 2 illustrates the accuracy drop in these models when some part of the data in a layer is lost. As seen, for large

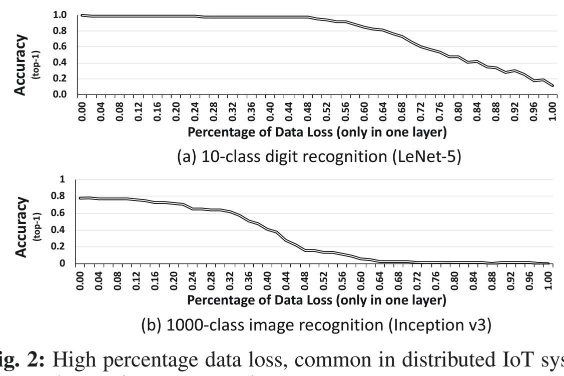


Fig. 2: High percentage data loss, common in distributed IoT systems, causes destructive accuracy drops.

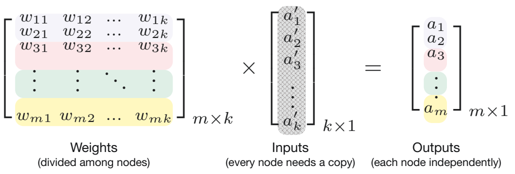


Fig. 3: Distribution of output splitting for FC layers.

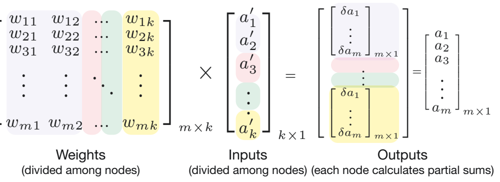


Fig. 4: Distribution of input splitting for FC layers.

percentages of data loss ($> 70\%$) per layer that are common in distributed IoT systems, the accuracy drop is destructive. Additionally, by comparing Figures 2a and b, we see that the sensitivity to data loss in more generalized models will only become worse. Since the amount of data loss happens in larger granularities, the current robustness methods in DNNs (e.g., bit-level tolerance) are insufficient to recover the loss. In contrast, our proposed robustness method is designed specifically for such a high amount of data loss and can recover from it with close-to-zero latency.

III. Robustness with CDC

This section first describes how distribution methods change computation, which helps us apply our robustness method at the library level. Next, we provide a simple example of our method and then we generalize it.

A. Distribution and Matrix Operations

Fully-Connected Layer: An FC layer l^{th} performs $\mathbf{a^l} = \sigma\big(\mathbf{W^l a^{l-1}} + \mathbf{b^l}\big)$, in which W, a, and b are weight, activation, and bias. First, we consider the GEMM: $\mathbf{W^l a^{l-1}}$. Figure 3 illustrates how output splitting affects weight and output matrices for an example with four devices. Since each device calculates a set of separate outputs, the output matrix is created separately by each device (and concatenated later). Such separation in output generation also divides the weight matrix along the y-axis, which has a one-by-one relationship with the output matrix division. Each device needs a copy of the input matrix, and the input matrix is not divided. In the input-splitting method, as Figure 4 depicts for the same four-devices example, the input matrix is divided between the devices. Similarly, the weights corresponding to those inputs are divided along the x-axis among devices. Each device calculates partial sums for all output elements. Finally, all partial sums are aggregated to create the final output. We can extend the above reasoning to bias and activation. For output splitting, biases, and the activation function can also be divided among the devices. But, for input splitting, both need to be applied after the aggregation. Since the majority of the computation time of DNNs is spent on GEMMs, such a difference does not have a big impact on computation time.

Convolution Layer: The channel-splitting method divides the filter weight matrix along the y-axis, as Figure 5 shows for two devices. Likewise, since the output is unrolled, such division translates to a similar along-the-y-axis division of the output matrix. Hence, channel splitting in convolution layers is the same as output splitting into FC layers and any robustness analysis is applicable on both, but with a different set of weights and inputs (i.e., unrolled version of filters and patches in convolution layers). In the spatial-splitting method, since each input patch is unrolled column-wise in the input matrix when we spatially divide the input, this division translates to an along-the-x-axis division of the input matrix. However, unlike input splitting in FC layers, filter weights cannot be divided. Therefore, spatial splitting, as conceptually shown in Figure 6 for two devices, divides the input matrix I with size $H{\times}W{\times}C$ in $\mathbf{O}_{K\times WH} = \mathbf{W}_{K\times F^2C} \times \mathbf{I}_{F^2C\times WH}$, in which C and K are the number of channels and filters, and filter size is $F{\times}F{\times}C$. In the filter-splitting method, a close representation of input splitting for FC layers, both filter weights and input are divided depth-wise. Since both filter weights and input are unrolled, we need to divide the weight and input matrices along the x- and y-axes, respectively. This distribution is similar to the outer product approach in matrix multiplication, versus the most commonly known algorithm of the inner product approach. Figure 7 shows this approach with two devices. Each device produces a partial sum for all elements. To create the final output, the final device aggregates all the elements and applies the activation function.

B. Robustness: A Simple Example

We propose a simple example of our CDC-based robustness to facilitate understanding. Consider an FC layer with two input and output elements, written as:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}. \tag{1}$$

Assume that we perform output splitting. We add a row to the weight matrix with the value of $[w_{11} + w_{21} \quad w_{12} + w_{22}]$ to create the summation of two outputs, or $a_1 + a_2$. With such an addition, the above equation becomes:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{11} + w_{21} & w_{12} + w_{22} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_1 + a_2 \end{bmatrix}. \tag{2}$$

As the summation of the weights can be done offline and is not dependent on inputs, we can rewrite the above equation as:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w^{cdc}_{:1} & w^{cdc}_{:2} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a^{cdc} \end{bmatrix}. \tag{3}$$

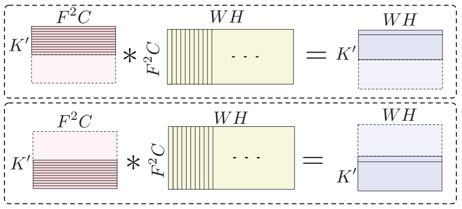


Fig. 5: Distribution of channel splitting for convolution layers.

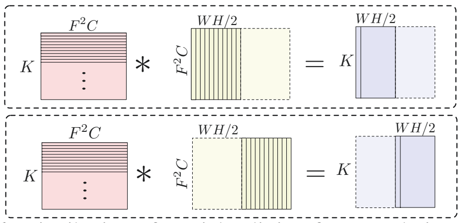


Fig. 6: Distribution of spatial splitting for convolution layers.

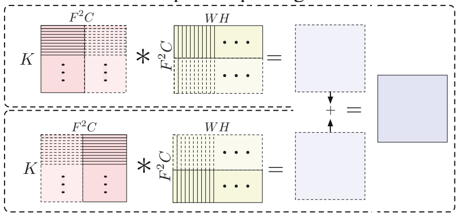


Fig. 7: Distribution of filter splitting for convolution layers.

The newly added weights to the weight matrix are the column sums of the weight matrix that is done offline before loading the weights. Therefore, with the addition of another device, we can guarantee to recover from one missing output with only a local subtraction in the final device. This method has three main benefits: **(i)** First, this level of guarantee on all devices is just with the addition of one device, compared to a double modular redundancy method that duplicates all devices. **(ii)** Second, this method is faster than redoing all operations since the subtraction of two local values that we already have received is almost more immediate than restarting all operations. This is because, the vanilla recovery method consists of loading a set of new weights (corresponding to the missing values) in the final device, asking for input from previous devices, and performing multiplications with all of its associated overhead of communication. **(iii)** Third, although we introduced the computations corresponding to a^{cdc}, these computations are similar in nature to the computations of a_1 and a_2. Hence, the distribution of these newly added computations follows the same rules and would not create an imbalance in the modified distribution.

C. Generalization of Robustness

This section extends our simple scenario, where each device computes only one output element, to a more realistic scenario, where each device computes hundreds of elements. Similarly, we showcase the output-splitting method as our example. Assume an FC layer performing the below equation:

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mk} \end{bmatrix}_{m\times k} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_k \end{bmatrix}_{k\times 1} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}_{m\times 1}. \tag{4}$$

By distributing the computations among two devices, each of the devices performs the computations for $m/2$ of output elements. The computations per each device are

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{\frac{m}{2}1} & w_{\frac{m}{2}3} & \dots & w_{\frac{m}{2}k} \end{bmatrix}_{\frac{m}{2}\times k} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_k \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{\frac{m}{2}} \end{bmatrix}, \text{and} \tag{5}$$

$$\begin{bmatrix} w_{(\frac{m}{2}+1)1} & w_{(\frac{m}{2}+1)2} & \cdots & w_{(\frac{m}{2}+1)k} \\ w_{(\frac{m}{2}+2)1} & w_{(\frac{m}{2}+2)2} & \cdots & w_{(\frac{m}{2}+2)k} \\ \vdots & \vdots & \vdots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mk} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_k \end{bmatrix} = \begin{bmatrix} a_{(\frac{m}{2}+1)} \\ a_{(\frac{m}{2}+2)} \\ \vdots \\ a_m \end{bmatrix} \tag{6}$$

in which input matrices are the same, but the weight matrix is divided along the y-axis. Each device creates separate parts of the output matrix. To introduce robustness, the new weight matrix would be as follows:

$$\begin{bmatrix} w_{11}+w_{(\frac{m}{2}+1)1} & w_{12}+w_{(\frac{m}{2}+1)2} & \cdots & w_{1k}+w_{(\frac{m}{2}+1)k} \\ w_{21}+w_{(\frac{m}{2}+2)1} & w_{22}+w_{(\frac{m}{2}+2)2} & \cdots & w_{2k}+w_{(\frac{m}{2}+2)k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{\frac{m}{2}1}+w_{m1} & w_{\frac{m}{2}2}+w_{m2} & \cdots & w_{\frac{m}{2}k}+w_{mk} \end{bmatrix}_{\frac{m}{2}\times k} . \tag{7}$$

By multiplying this new weight matrix with inputs, the below output matrix is created:

$$\begin{bmatrix} a_1 + a_{(\frac{m}{2}+1)} \\ a_2 + a_{(\frac{m}{2}+2)} \\ \vdots \\ a_{\frac{m}{2}} + a_m \end{bmatrix}_{\frac{m}{2}\times 1} , \tag{8}$$

which is is the summation of two output matrices in Equations 5 and 6. Therefore, by introducing such a weight matrix as Equation 7, we can introduce robustness. Similar to our simple example, the computation of new weights is done offline, recovery has a close-to-zero latency, the robustness covers all devices, and the new computation is balanced. In contrast, splitting methods that divide the input matrix among the devices do not yield similar benefits. To illustrate why, we study input splitting among two devices for the computation of the FC layer in Equation 4. Input splitting for FC layers divides the input and the weight matrix along the x-axis. Accordingly, the computations per device are from Equations 9 and 10.

Each of these equations calculates a partial sum. However, as seen, no share factor exists between the two computations. Therefore, to perform coded distribution, a third device needs to perform the entire calculations of Equations 9 and 10, which creates unbalanced work between the devices and has no advantage over just replicating the entire work as modular redundancy methods do.

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1\frac{k}{2}} \\ w_{21} & w_{22} & \cdots & w_{2\frac{k}{2}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{m\frac{k}{2}} \end{bmatrix}_{m\times\frac{k}{2}} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_{\frac{k}{2}} \end{bmatrix}_{\frac{k}{2}\times 1} = \begin{bmatrix} \delta a_1 \\ \delta a_2 \\ \vdots \\ \delta a_m \end{bmatrix}_{m\times 1} \tag{9}$$

$$\begin{bmatrix} w_{1(\frac{k}{2}+1)} & w_{1(\frac{k}{2}+2)} & \cdots & w_{1k} \\ w_{2(\frac{k}{2}+1)} & w_{2(\frac{k}{2}+2)} & \cdots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m(\frac{k}{2}+1)} & w_{m(\frac{k}{2}+2)} & \cdots & w_{mk} \end{bmatrix}_{m\times\frac{k}{2}} \times \begin{bmatrix} a'_{\frac{k}{2}+1} \\ a'_{\frac{k}{2}+2} \\ \vdots \\ a'_k \end{bmatrix}_{\frac{k}{2}\times 1} = \begin{bmatrix} \delta a_1 \\ \delta a_2 \\ \vdots \\ \delta a_m \end{bmatrix}_{m\times 1} \tag{10}$$

Distribution Techniques Suitable for Robustness: Based on our discussion, among the distribution methods for DNNs, only some are suitable for our CDC-based robustness. Such suitable methods do not split the input elements but split the weights. Table I provides a summary of all the presented methods and whether they are suitable for robustness. For FC layers, the output-splitting method is suitable for robustness.

TABLE I: Distribution Techniques Suitable for Robustness.

Layer	Distribution Method	Divides Input	Divides Weight	Divides Output	Suitable for Robustness
`fc`	Output	✗	✓	✓	Yes
	Input	✓	✓	✗	No
`conv`	Channel	✗	✓	✓	Yes
	Spatial	✓	✗	✓	No
	Filter	✓	✓	✓	No

For convolution layers, the channel-splitting method has similar characteristics. Unfortunately, the rest of the distribution methods are not suitable for robustness because to introduce robustness, they need to perform the entire computation again, which causes communication overhead. For instance, in spatial splitting, although every device has all the weights, they only own some part of the input. Therefore, with our technique, we need another device to perform the computation based on the summation of the input parts. Since input elements change, computing such a summation has an overhead during the runtime (2x compute). The filter-splitting method also suffers from the fact that no element from the input or weights is shared between computing devices.

IV. Experiments

Experiments Setup: We evaluate our method on a distributed system with RPi with 1.2 GHz Quad Core ARM Cortex-A53 CPU and a 900 MHz 1 GB RAM LPDDR2 memory. We choose RPi because they represent the de facto choice for several IoT systems, they are readily available, and they allow common software packages. Our implementation is created with a software stack based on Docker containers. We use Keras 2.1 with the TensorFlow backend (version 1.5). For RPC calls and serialization, we use Apache Avro. A local WiFi network with a measured bandwidth of 94.1 Mbps and a measured client-to-client latency of 0.3 ms for 64 B is used.

Task Creation & Assignment: The policy of task creation in IoT-based distributed DNN systems is done with either profiling or heuristics that use common monitoring/managing tools such as Kubernetes. The policies create tasks per device for a given DNN by studying its memory footprint, computation requirement, and communication overhead. Regardless of the policy that finds the optimal distribution (out of the scope of this paper, see [4]), all the pre-trained weights are loaded to each device storage so that a device can switch its assigned task easily if needed. For each number of available devices, a single task allocation file is loaded to all devices and each device performs its allocated tasks based on the file. We use an IP table file to assign tasks to each RPi. CDC weights are also created offline and loaded to the storage. In the case of a failure, the system uses another pre-defined distribution file with fewer devices that has a lower performance. In such a case, since the detection of a missing device takes time, the system mishandles many requests. Our proposed solution has tolerance to such failures, so the system never loses a request. Additionally, with a close-to-zero recovery latency, the system proactively is more tolerant to straggler nodes.

Weight Storage: Each Pi has an SD card storage, for storing the weights, which is relatively inexpensive compared to the

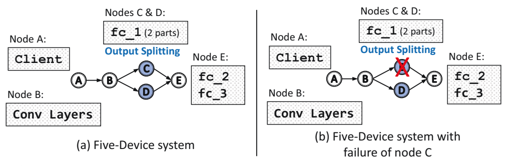


Fig. 8: Case study I: AlexNet on a five-device system.

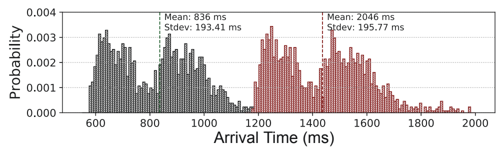


Fig. 9: Case study I: Recovery latency with & without CDC.

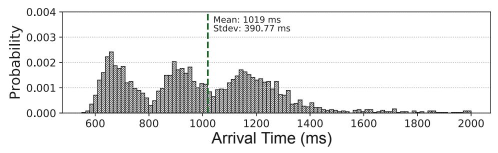


Fig. 11: AlexNet latency histogram without straggler mitigation.

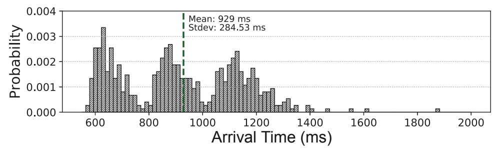


Fig. 12: AlexNet latency histogram with straggler mitigation.

main memory. All trained weights are loaded to each Pi's storage (16 GB storage in our system), so each Pi can be assigned to execute any part of a layer. If local storage is limited, the assigned weight can also be shared on the network from a network-storage filesystem. This approach makes a tradeoff between how fast the switching time between different models can be and per-device storage usage. Additionally, note that the distribution method does not replace other methods, such as offloading to servers. The decision is on a per-case basis and depends on several system-level decisions. The distribution offers the additional option of processing data locally.

A. System Recovery Case Studies

Case Study I: To depict the impact of how failures affect a system, we deploy AlexNet [17] on two IoT systems. The first system (Figure 8a) contains five devices. The first fully-connected layer is split with the output-splitting between two devices with no robustness method. The black bars in Figure 9 show the latency of the system when performing single-batch inferences. If device C experiences failure, as shown in Figure 8b, other devices need to perform the task assigned to the failed device. Since device C computes half of the first fully-connected layer, device D needs to perform this extra task. After the failure is detected, which takes tens of seconds, the red bars in Figure 9 depict the new shifted latency histogram of the system. Based on our measurements, on average, the system experiences 2.4x slowdown after recovery. The system does not perform beneficial work during failure detection and experiences a significant slowdown afterward. However, with our method, the system does not experience any slowdown or service interruption.

Case Study II: As a remedy to failures, we deploy AlexNet on a six-device system. Figure 10a shows this system, in which an extra device is added for robustness using CDC. Note that our goal is to create robustness only for the first fully-connected layer and the extra device provides robustness to all the computations done on devices D and E. If we experience failure (Figure 10b), the performance of the system does not change. Additionally, during the operation without failure, we use the extra device to mitigate the straggler problem. Figures 11 and 12 show the system latency with and without this mitigation, respectively. As shown, the range and the distribution of latencies are improved towards a better performance. Thus, in addition to robustness, we can exploit the extra device to increase the performance.

B. Straggler Mitigation

We study straggler mitigation benefits by extending the previous system. To initiate recovery, a device waits for a particular amount of time. By adjusting this waiting threshold in a device, we can treat our method as a solution for the straggler problem after receiving the necessary amount of data. A lower threshold reduces latency and thus increases performance. The straggler problem is more prominent with more devices, so we set up an experiment for a system with four devices, each of which performs a split in a fully-connected layer (Figure 13a). Figure 13b shows performance improvement of straggler mitigation with a diffident number of devices. The performance improvement is compared with the same system, with the same number of devices, and with no straggler mitigation. As seen, for more devices, straggler mitigation has better performance (up to 35%) compared with a no-straggler-mitigation system with the same number of devices.

C. Full Model Coverage

In the system in Figure 10, devices with model parallelism are robust with CDC. For other devices, by replicating the device's task (N-modular redundancy with $N = 2$, or 2MR), we can tolerate one failure. A hybrid approach (CDC+2MR) can cover the system for failures. Our method covers any number of devices in one layer with just one additional device (for

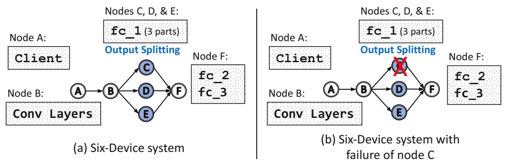


Fig. 10: Case Study II: AlexNet on a six-device system.

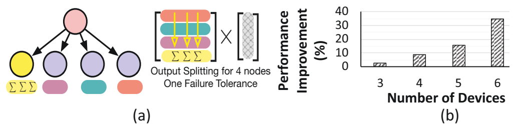


Fig. 13: Straggler mitigation study. (a) A system setup for four devices. (b) Straggler mitigation performance.

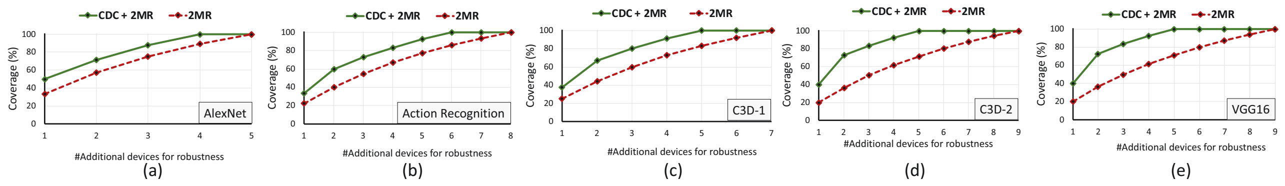


Fig. 14: Full model coverage studies.

robustness to one failure). However, 2MR requires an additional device for each device, resulting in a linear number of additional devices. Our method has a *constant* cost with an additional number of devices, whereas 2MR requires a *linear* number of additional devices. Figure 14 examines several DNNs [18]–[20] with distributed implementations, showcasing tolerance to one failure with 2MR-only and CDC+2MR. Since CDC requires fewer devices than 2MR to cover devices with model parallelism, the number of additional devices for full coverage in CDC+2MR is smaller than that of 2MR. The amount of difference depends on the number of layers distributed with model parallelism and the number of devices used per layer. For example, Figure 14c and d depict two C3D distributions with different numbers of devices for the layers utilizing model parallelism (two vs. three devices). In Figures 14c and d, with two additional devices, CDC+2MR achieves coverage of 67% and 73% compared to 2MR's coverage of 44% and 36%. This is because C3D distribution has two layers with model parallelism. Therefore, CDC+2MR achieves better coverage compared to 2MR. To summarize, by utilizing model parallelism for a layer with N devices, our method allows hiding a single node's failure at a cost of $(1+\frac{1}{N})$ times the hardware cost, as opposed to 2MR's cost of 2x hardware.

V. Discussions

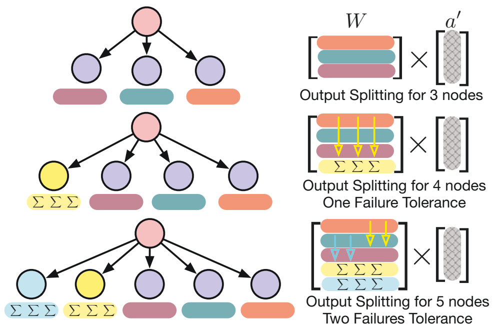
The Introduced Computation: The introduced new computations for our CDC-based method are similar to that of underlying GEMM computations. This is because the newly added weights are added to the weight matrix and can be calculated without the user's input at the library level. Therefore, there are no additional costs for reprogramming the applications. Moreover, since the nature of the computations for these new weights is similar to that of DNNs, there is no need to design new kernels or distribution methods.

Extending Robustness to More Failures: Our discussions focused on tolerating up to one failure. However, Extending to more than one failure is possible by adding new devices that compute based on the summation of some rows of weights instead of all. Figure 15 illustrates three setups in order of increasing tolerance to failures. The last setup tolerates two failures because new devices perform partial sums on the weights.[1] Thus, by utilizing idle devices with an overlapping set of weights, the system's robustness increases.

VI. Related Work

CDC [13] introduces coding for MapReduce-type workloads for large-scale computing. By coding, which increases the

[1] Note that the coverage to two failures is almost complete (partial error correction). We need Hamming-style coverage for full error correction.



W
a'
$\Sigma\ \Sigma\ \Sigma$
Output Splitting for 3 nodes
Output Splitting for 4 nodes
One Failure Tolerance
Output Splitting for 5 nodes
Two Failures Tolerance

Fig. 15: Tolerating multiple failures.

computation of mapping functions, communication can be reduced in the reduction phase. The authors of the CDC theoretically study the limits and trade-offs of such distribution and illustrate an inverse relationship between the amount of computation and communication. Usually, coding in CDC is applied over bit-level representation of numbers. *Instead of coding over floats/bits, we apply coding to the application level by introducing new weights. Furthermore, in contrast, to reduce communication overhead in other studies, our goal is to increase robustness and tolerate unstable latencies.* CDC helps mitigate the straggler problem in computing clusters [21], [22], besides other methods such as straggler detection algorithms [23], [24] and replication-based approaches [25], [26]. Several studies also utilize CDC to mitigate the straggler problem in distributed storage systems [27]. Distributed learning algorithms have also used CDC opportunities [14]. Since these algorithms use data parallelism for learning, CDC facilitates the mapping phase in learning algorithms with data shuffling. Particularly, Lee et al. [14] focused on two basic blocks of learning algorithms, matrix multiplication, and data shuffling. *None of the above works has studied CDC in the context of robustness.* In contrast with our work, distributed learning studies [14] examine large-scale learning algorithms, which employ data parallelism, *whereas our work focuses on IoT-based inferencing, which utilizes model parallelism.*

VII. Conclusion

We proposed a method to introduce tolerance for the single-batch inferencing of DNNs, a key operation in IoT. Our method exploits model-parallelism methods in prevalent DNN layers to add balanced computation for robustness. Model parallelism helps us achieve efficient system distribution. We extended CDC to provide a trade-off between computations and robustness on distributed IoT.

References

[1] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *2017 Design, automation and Test in eurpe (Date)*. IEEE, 2017, pp. 1396–1401.

[2] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 328–339.

[3] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–2.

[4] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Towards collaborative inferencing of deep neural networks on internet of things devices," *IEEE Internet of Things Journal*, 2020.

[5] B. Asgari, R. Hadidi, J. Dierberger, C. Steinichen, A. Marfatia, and H. Kim, "Copernicus: Characterizing the performance implications of compression formats used in sparse workloads," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 1–12.

[6] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 253–266.

[7] B. Asgari, R. Hadidi, N. S. Ghaleshahi, and H. Kim, "Pisces: power-aware implementation of slam by customizing efficient sparse algebra," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[8] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018, pp. 115–127.

[9] B. Asgari, R. Hadidi, and H. Kim, "Ascella: Accelerating sparse computation by enabling stream accesses to memory," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2020, pp. 318–321.

[10] D. Pena, A. Forembski, X. Xu, and D. Moloney, "Benchmarking of cnns for low-cost, low-power robotics applications," in *RSS 2017 Workshop: New Frontier for Deep Learning in Robotics*, 2017.

[11] X. Zhang, Y. Wang, and W. Shi, "pcamp: Performance comparison of machine learning packages on the edges," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[12] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," *International Conference on Machine Learning (ICML'18)*, 2018.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations*. ACM, 2015.

[13] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, Jan 2018.

[14] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, March 2018.

[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *26th Annual Conference on Neural Information Processing Systems (NIPS)*. ACM, 2012, pp. 1097–1105.

[18] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters (RA-L), Invited to IEEE/RSJ International Conference on Intelligent Robots and Systems 2018 (IROS)*, vol. 3, no. 4, pp. 3709–3716, Oct 2018.

[19] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning Spatiotemporal Features with 3D Convolutional Networks," in *Computer Vision (ICCV), 2015 IEEE International Conference on*. IEEE, 2015, pp. 4489–4497.

[21] N. S. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*. IEEE, 2016, pp. 954–960.

[22] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Advances In Neural Information Processing Systems*, 2016, pp. 2100–2108.

[23] R. Bitar, P. Parag, and S. El Rouayheb, "Minimizing latency for secure distributed computing," in *Information Theory (ISIT), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 2900–2904.

[24] K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Coded computation for multicore setups," in *Information Theory (ISIT), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 2413–2417.

[25] D. Wang, G. Joshi, and G. Wornell, "Efficient task replication for fast response times in parallel computation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 599–600.

[26] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" *IEEE Transactions on Communications*, vol. 64, no. 2, pp. 715–722, 2016.

[27] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes can reduce queueing delay in data centers," in *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 2766–2770.