INTEGRATION, the VLSI journal 100 (2025) 102299

Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

Model and system robustness in distributed CNN inference at the edge

Xiaotian Guo [a,b,*], Quan Jiang [c], Andy D. Pimentel [a], Todor Stefanov [b]

[a] Informatics Institute, University of Amsterdam, Amsterdam, 1098 XH, The Netherlands
[b] Leiden Institute of Advanced Computer Science, Leiden University, Leiden, 2333 CC, The Netherlands
[c] Informatics Institute, Nanjing Agricultural University, Nanjing, 210095, China

ARTICLE INFO

Keywords:
Embedded systems
System resilience
Fault tolerance
Distributed systems
Deep learning

ABSTRACT

Prevalent large CNN models pose a significant challenge in terms of computing resources for resource-constrained devices at the Edge. Distributing the computations and coefficients over multiple edge devices collaboratively has been well studied but these works generally do not consider the presence of device failures (e.g., due to temporary connectivity issues, overload, discharged battery of edge devices). Such unpredictable failures can compromise the reliability of edge devices, inhibiting the proper execution of distributed CNN inference. In this paper, we present a novel partitioning method, called RobustDiCE, for robust distribution and inference of CNN models over multiple edge devices. Our method can tolerate intermittent and permanent device failures in a distributed system at the Edge, offering a tunable trade-off between robustness (i.e., retaining model accuracy after failures) and resource utilization. We verify the system's robustness by validating the overall end-to-end latency under failures. We evaluate RobustDiCE using the ImageNet-1K dataset on several representative CNN models under various device failure scenarios and compare it with several state-of-the-art partitioning methods as well as an optimal robustness approach (i.e., full neuron replication). In addition, we demonstrate RobustDiCE's advantages in terms of memory usage and energy consumption per device, and system throughput for various system setups with different device counts.

1. Introduction

As Artificial Intelligence (AI) continues its rapid evolution, convolutional neural networks (CNNs) are becoming increasingly prevalent across a variety of applications [1]. The surge of Internet-of-Things (IoT) devices has also elevated the deployment requirements of CNNs at the Edge. However, the growing complexity and size of CNN models, such as VGG-16 [2], and CoAtNet-6 [3], pose a significant challenge in terms of computing resources for resource-constrained edge devices. One approach to address this challenge is to construct a lightweight CNN model from a large CNN model utilizing model compression [4] or neural architecture search [5] which may decrease accuracy. Another approach is to distribute a large CNN model between edge devices and cloud servers [6], but this approach introduces unpredictable inference latency and raises trustworthiness, security, and privacy concerns.

To address these issues, studies on fully distributing the CNN inference over multiple edge devices have been proposed without the need for model compression and cloud servers. In such a *planar CNN distribution paradigm*, model partitioning [7,8] and data partitioning [9,10] methods are typically applied to alleviate the discrepancy between the constrained resources of edge devices and the huge requirements of deploying large CNN models. However, these partitioning methods assume continuous availability of all involved edge devices that cannot be always guaranteed because an edge device could be temporarily unreachable (especially when edge devices are mobile and use low-power short distance radios for communication) or a device could experience a temporary failure (e.g., due to a discharged battery). Therefore, it is imperative to devise and utilize partitioning methods for distributed CNN inference with robustness in mind.

In this paper, we present a novel partitioning method, called **RobustDiCE**, for robust distribution and inference of CNN models over multiple edge devices. RobustDiCE features both *system robustness*, i.e., CNN inference can continue execution even if one or more edge devices fail to function properly, and *model robustness*, i.e., preserving the inference accuracy of the CNN model as much as possible when some of the intermediate CNN inference results are lost due to failed devices. We improve the system robustness by implementing a decentralized connections between multiple devices where each device incorporates a robust fault handler for reliable execution. The fault handler's internal timeout mechanism, regulated by periodic heartbeats, prevents system deadlocks and improves resilience against potential device failures or network disruptions. Moreover, we address the model robustness challenge by evaluating the relative importance of each neuron in

* Corresponding author at: Informatics Institute, University of Amsterdam, Amsterdam, 1098 XH, The Netherlands.
E-mail address: gxtzhuxi@gmail.com (X. Guo).

https://doi.org/10.1016/j.vlsi.2024.102299
Received 31 May 2024; Received in revised form 4 September 2024; Accepted 12 October 2024
Available online 20 October 2024

the CNN model and then partitioning these different neurons of each CNN layer into different groups (to be mapped to the various edge devices) as 'evenly' as possible. Our main novel contributions can be summarized as follows:

- Based on the importance criterion of different neurons in each CNN layer, a new partitioning method is proposed to preserve the model accuracy as much as possible against device failures. This new method combines *partial* neuron replication and *importance-aware* neuron clustering to achieve CNN model robustness. It also provides a tunable trade-off between robustness (i.e., retaining model accuracy after failures) and resource utilization.
- We evaluate our novel partitioning method on several representative CNN models using the ImageNet-1K dataset under pessimistic device failure scenarios, as enabled by RobustDiCE's system robustness support. We compare it with a number of state-of-the-art (partitioning) approaches, including the CDC method [11] leveraging neuron replication to increase robustness, and an ideal robustness approach utilizing full neuron replication.
- We demonstrate our method's superiority in terms of memory usage and energy consumption per device, and system throughput under different system configurations.

The remainder of the paper is organized as follows. Section 2 discusses related work, after which Section 3 gives a simple motivational example illustrating the need for our novel partitioning method RobustDiCE. Section 4 presents the RobustDiCE method in detail. In Section 5, we describe a range of experiments, demonstrating the merits and advantages of RobustDiCE in terms of system and model robustness as well as resource utilization. Finally, Section 6 concludes the paper and discusses the limitations of our work as well as possible future directions of research.

2. Related work

Model and data partitioning methods [7,12] can be exploited to distribute the workload of a large CNN model inference along the edge–cloud continuum or fully among multiple edge devices, thus reducing the required computation resources of edge devices [8]. Considering the edge–cloud continuum, studies [7,10,13–16] distribute the workload of a large CNN model in a planar fashion over multiple devices at the edge without using the cloud. Processing the input data collaboratively by utilizing multiple edge devices helps to mitigate the resource limitations of a single edge device in terms of available memory, energy budget, etc. As mentioned in Section 1, there are two common methods for implementing planar distributed CNN inference: model partitioning and data partitioning.

Data partitioning involves dividing the input data into smaller chunks, with each chunk processed on a separate device. For example, DeepThings [13] uses the Fused Tile Partitioning (FTP) method for splitting input data frames of CNN layers in a grid fashion to reduce the CNN memory usage per device. Alternatively, the model partitioning method splits the CNN layers and/or connections of a large CNN model, thereby creating several smaller sub-models (model partitions) where each sub-model is executed on a different edge device. For example, MoDNN [10] and DeeperThings [7] split the neurons in convolution layers and fully connect layers of the VGG-16 model to reduce layer computations per device. In [12,15,16], CNN layer connections are split and each CNN layer is treated as a sub-task. These sub-tasks are then mapped to edge devices through a balanced processing pipeline approach. In addition to using data and model partitioning to map large CNNs on resource-constrained edge devices, researchers try to optimize the CNN partitioning to improve inference performance under different conditions such as network bandwidth, neural network topology, and hardware specifications [8]. For example, the methodologies in [9,16–18] propose algorithms to determine partitioning policies that generate efficient CNN mappings to improve the performance of cooperative inference over multiple edge devices. [8] explores specific optimal CNN partitionings to reduce memory usage and energy consumption per device. However, these partitioning methods assume that the involved computing devices/servers (and communication links) between them are always available and work properly. Our partitioning method is designed to be robust against temporary or permanent failures of devices.

System robustness in distributed CNN inference refers to the ability of a system to continue functioning correctly, or to provide graceful degradation, even in the presence of hardware or software failures (such as the unavailability of system resources, communication failures, invalid or excessive input data, etc.) [19]. The majority of studies on robust computing focus on replication [20–22], redundancy [23,24], error correction [11,25], checkpoint recovery [26], and fault tolerance techniques [27]. For example, [20–22] keep the results of multiple computing nodes up to date and consistent through replication. [23,24] replicate computing nodes of the distributed system to provide multiple identical instances as backups in case of a node failure. However, full replication of input data and CNN layers (with their vast numbers of weights/biases) for CNN inference is not feasible for resource-constrained edge devices. Moreover, the usage of many redundant hardware devices is expensive and not suitable in all cases. [25] uses error correcting codes (ECCs) to protect the weights from perturbations while [11] applies a coded distributed computing (CDC) method for fast recovery of output results from a certain number of node failures. However, error correction methods introduce extra computations on the edge devices which may be difficult to facilitate given the limited available resources. [26] provides a checkpoint recovery mechanism for "continued execution" where the deep learning implementation continues to execute by utilizing the remaining set of computing nodes. However, the recovery method increases the physical memory usage, and the recovery time is limited to the filesystem I/O bottleneck of the edge devices. Unlike the previous works, our work focuses on robust, distributed CNN inference with the goal of reducing the computational and memory resource usage per edge device to better match the limited resources of edge devices.

Model robustness of distributed CNN inference concerns the property of a model of being resilient in terms of inference accuracy to the failure of physical computing nodes due to power outages, unstable inter-node connections, other hardware/software failures, etc. In distributed CNN inference, the missing neurons on those failed nodes may result in a significant accuracy drop [11] of a CNN model. The code distributed computing (CDC) method in [11] utilizes one additional, presumed functional device to back up the summation of partitioned neurons of other distributed devices and use that to recover the output of missing neurons in the event of a single node failure. Our method, on the other hand, can cope with multiple node failures without integrating additional devices and computations. To alleviate the influence of node failures on the CNN inference accuracy, several failure-aware retraining methods [28–33] for CNNs have been developed. For example, if layer connections of CNN models are split, the forward process of the CNN inference cannot continue because of the presence of failed nodes. DeepFogGuard [34] establishes skip hyperconnections to skip certain failed physical nodes during the retraining process. ResiliNet [31] introduces failout to simulate physical node failure conditions during retraining. [30] retrains CNN models to be resilient to packet loss in a lossy IoT network. The retraining process adds dropout on certain CNN layers but this cannot guarantee that the model accuracy is preserved. When the dropout rate is too high, thereby simulating a high percentage of node failures, the retraining model may result in under-learning which causes a significant decrease in its accuracy. In addition, all these retrained models are designed to be aware of only specific failures such as communication failures between two CNN layers, certain node failures in a pipeline multi-node inference, etc. Moreover, CNN retraining requires a large amount

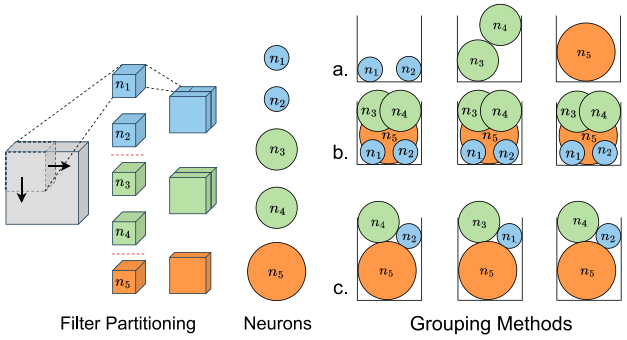


Fig. 1. Typical vs. Robust partitioning.

of data that may not be always accessible for an end user of a pre-trained CNN model to perform retraining before the deployment in an unreliable environment. As most state-of-the-art pre-trained models directly available to an end user for deployment are not failure-aware, our RobustDiCE method can be easily applied to partition these pre-trained models to achieve system and model robustness without any retraining, without assuming specific types of failures, and without suffering from accuracy degradation due to parameter changes (e.g., by adding dropout on CNN layers) in the neural networks. Moreover, our robustness method can be seen as complementary to these retraining approaches, i.e., if we would apply our method to the aforementioned retrained models, we can further improve their robustness against node failures.

To summarize, performing robust inference on distributed edge devices is vital. Existing robustness methods suffer from extra computing resource requirements, time-consuming retraining, accuracy degradation, etc. In contrast, our method RobustDiCE is designed to guarantee robustness under limited resources of edge devices. In addition, our method is a post-training technique that provides robustness without the need for CNN model retraining. Furthermore, we have implemented and tested our robust distributed CNN inference on real physical edge devices. Both system robustness and model robustness are provided by our method and verified via experimental results.

3. Background and motivation

In this section, we provide some background information and a motivational example to understand our novel CNN partitioning method for robustness.

In general, state-of-the-art partitioning methods, such as discussed in [7], do not consider robustness as they do not consider the fact that different neurons/filters in CNN layers have different *importance*, thereby causing various effects on the inference accuracy of a CNN model, particularly those neurons with larger values [35]. The relative *importance* of a neuron in a CNN layer can be measured by calculating metrics such as the l_1-norm [36] and l_2-norm [37], just to name a few. 'To partition a CNN layer with robustness in mind, it is essential to find an effective way to group and distribute its neurons/filters over computing nodes as evenly as possible in terms of *importance*.

To clarify this statement, we use the simple example shown in Fig. 1, where we consider a convolution layer with five filters/neurons denoted as n_1 to n_5. We want to partition these neurons over three computing nodes. In this example, the importance score s_j of each neuron n_j is measured by calculating the l_1-norm. That is, taking the filter corresponding to neuron n_j with shape $C_{in}\times k\times k$ (where k denotes the kernel size of the filter and C_{in} the number of input channels), we calculate the sum of absolute values of all the weights in the filter and its bias using the magnitude-based method l_1-norm. In Line 7, $W_j^{c,h,w}$ denotes a particular weight value in the jth filter corresponding to neuron n_j, and b_j its bias. In the middle and the right part of Fig. 1, we visualize the importance s_j of each neuron n_j by the size of the circle representing the neuron, i.e., neuron n_5 has the highest importance whereas n_1 and n_2 have the lowest importance.

As shown in Fig. 1(a), a partitioning method without robustness in mind (i.e., no consideration of the neurons' importance s_j) splits the five neurons into three groups (visualized by the three colors in Fig. 1) and the groups are distributed over the three nodes. Such distribution reduces computational resources per node because the layer workload is split over the nodes. However, this distribution is not robust at all because if, for example, the third node fails, which runs the most important neuron n_5, then the inference accuracy will decrease significantly.

To maximize the robustness, well-known modular redundancy methods can be applied as shown in Fig. 1(b). Here, we replicate all neurons over the three nodes, thereby achieving maximum robustness against failures because even if one or two nodes fail then the remaining available node will run all the neurons without a decrease in inference accuracy. However, this significantly increases the resource requirements (e.g., memory and energy consumption) for each node. Moreover, this full replication approach might be infeasible for resource-constrained nodes due to the limitations with respect to their computational or memory resources and the possible energy budget of an edge device.

The two example scenarios, illustrated in Fig. 1(a) and (b), clearly show that using existing, robustness-unaware partitioning methods or modular redundancy methods in isolation cannot provide efficient and robust distributed CNN inference on multiple resource-constrained edge devices. Therefore, in this paper, we propose a novel method, explained in detail in Section 4, which *combines replication and importance-aware partitioning* to achieve high and tunable robustness in an efficient way for distributed CNN inference. The result of applying our method to our simple example is illustrated in Fig. 1(c). The basic idea is that some (not all) neurons in a CNN layer are replicated and all neurons (initial and replicas) are partitioned into groups and distributed evenly over the nodes based on their *importance*.

The advantage of this partitioning method is that if either the first or third node fails, the remaining nodes can still run all the neurons, preserving inference accuracy. If the second node fails, the critical neuron n_5 still remains, limiting the accuracy degradation. Therefore, we can achieve comparable robustness to the scenario in Fig. 1(b), but with reduced computational resource requirements, as not all neurons are replicated or run on each node.

4. The RobustDiCE method

Our method RobustDiCE features both *system robustness*, i.e., CNN inference can continue to execute or recover even if one or more computing nodes and/or communication links between them fail to function properly and *model robustness*, i.e., when some of the CNN neurons and/or their input/output data are lost due to the failed nodes/links, the inference accuracy of the CNN model is preserved as much as possible. In Section 4.1, we outline our decentralized

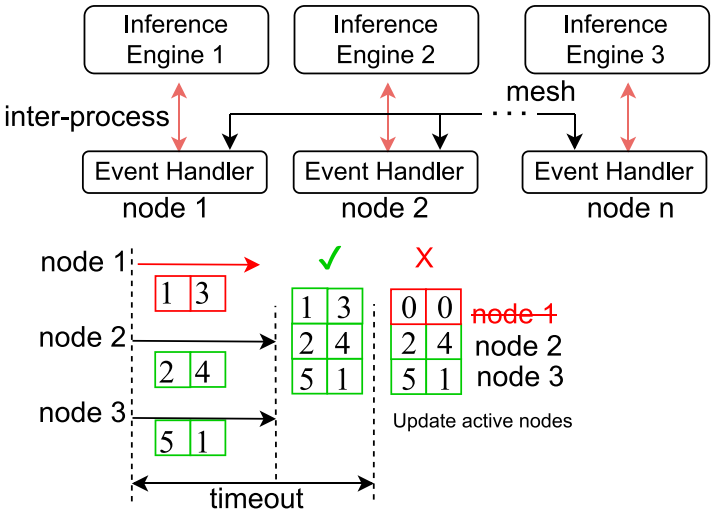


Fig. 2. Decentralized computing framework.

computing framework for CNN inference that supports system robustness. In addition, we support model robustness by applying our new partitioning method, presented in Sections Section 4.2, that combines partial neuron replication and importance-aware neuron grouping and distribution over multiple nodes.

4.1. Decentralized computing framework

In our work, we have devised and implemented a decentralized framework for distributed computing and communication. Such an infrastructure is crucial to ensure that the distributed CNN inference can be executed collaboratively and properly even in the case of node/link failures. In our decentralized framework, each node operates independently in an interconnected network, which simplifies management and enhances scalability without relying on centralized managing nodes. This mechanism offers significant advantages in terms of system robustness and reliability. For example, if one node fails, the other nodes autonomously detect and manage it, adjusting their operations until the failed node recovers and is reintegrated into the network. This contrasts with centralized systems, where a single point of failure in a management node can lead to disruption of the entire system.

A high-level overview of our decentralized computing framework for distributed CNN inference is shown in Fig. 2. In this framework, all nodes are interconnected in a peer-to-peer fashion through an N-to-N mesh topology. On each computing node, there are two running processes: *inference engine* and *event handler*. The inference engine mainly takes care of the CNN computations, whereas the event handler is responsible for handling events such as interconnections, heartbeat [38] and data synchronization between nodes, node failures, node management, etc. The two processes communicate with each other directly through the inter-process communication mechanism. As communication between nodes happens asynchronously with computation, a node can continue with its computation even if the communication with other nodes is not completed. This prevents the communication from becoming a bottleneck.

In the N-to-N mesh topology, each node acts as both a client and a server, thereby allowing for direct communication between nodes without centralized servers or coordinators. The client on each node connects with a set of servers on other nodes, and the server on each node is capable of replacing other servers if necessary. This design ensures that a failure of a node or communication link will not disrupt the entire system. The client in the event handler of each node continuously sends regular messages, known as heartbeats, to servers on other nodes to confirm their operational status. As shown in Fig. 2, we typically set a relatively large value for the timeout to ensure that the data synchronization between the event handlers of these nodes is completed before reaching the timeout. Once data synchronization is completed, the event handler will activate the inference engine to proceed with the corresponding computations. However, if a node fails to send out a heartbeat within a predefined timeout, other nodes assume that this unresponsive node has failed. Then, the server would remove the inactive or failed nodes and update its list of active nodes.

For example, consider the bottom part of Fig. 2 illustrating the timeout mechanism concerning the three nodes. When all nodes have completed their assigned computations, the partial results, i.e., the values in the green boxes on each node, need to be synchronized. If data communication proceeds without any failures, all nodes will receive the correct full results, as shown in the 2×3 green box configuration. However, if Node 1 encounters a failure and cannot send its partial results (values 1 and 3) to the other nodes, these nodes will end up waiting for Node 1. If Node 1 is unable to send a heartbeat within the predefined timeout period due to its failure, the event handlers on the other nodes will detect this failure and disconnect from Node 1.

Subsequently, all nodes, except Node 1, will proceed to zero out the vector content related to Node 1's contributions, indicated by the red box. Although the data synchronization results on these nodes will be incorrect without the input from Node 1, these nodes will still proceed and activate their inference engines to continue processing tasks. It should be noted that our decentralized framework ensures that no node is permanently eliminated from the system upon failure. Instead, we employ a timeout mechanism to manage transient faults during the inference process. This allows the system to temporarily bypass any faulty node and continue operations. In cases where a node fails to send a heartbeat within the predefined timeout, it is temporarily excluded from computations but can rejoin once it becomes operational again. For example, if a node recovers after a failure, it reintegrates into the system seamlessly due to our N-to-N mesh topology. As each node acts as both client and server, all working nodes would update their registry of active nodes to add the recovered node back to their communication list again. Nodes continuously exchange heartbeats (using a sufficiently large timeout value) to monitor each other's status and facilitate data synchronization before a node is considered unresponsive. This mechanism ensures that our system can adaptively maintain functionality even in the presence of node failures, without significant degradation in performance.

By enabling each node to perform its computation and communication independently, our decentralized framework offers robust and scalable computation and communication for distributed CNN inference. This approach ensures that the system remains functional and efficient even in the face of individual node or link failures.

4.2. Robust partitioning

In this section, we present our new partitioning method which achieves CNN model robustness by combining importance aware neuron grouping/clustering with partial neuron replication in order to evenly distribute the neurons in a CNN model over multiple nodes. We employ the so-called horizontal partitioning method, which is applied layer-wise on every layer until the whole CNN model is partitioned. The motivation to employ layer-wise horizontal partitioning across every layer of the CNN model stems from a strategic aim to enhance system robustness and fault tolerance. In distributed CNN architectures, partitioning can typically be approached either horizontally or vertically. Vertical partitioning distributes entire and different layers onto different devices, which can indeed improve throughput by allowing layers to be processed in a pipelined manner. However, this method exposes the system to significant risks; if one or more devices fail, entire layers of the CNN model are compromised, which can drastically impair the entire inference process. In contrast, our method involves horizontal partitioning, where each layer is divided and distributed across multiple devices. This approach ensures that the failure of a single device impacts only a part of each layer rather than entire layers, thus allowing for partial recovery and continuation of the inference process with minimal disruption.

The general layer-wise partitioning procedure is outlined in Algorithm 1. It accepts as inputs a set of computational layers L from the CNN model and their coefficients W as well as the total number of computing nodes ND across which the CNN model will be distributed. Additionally, a set T of threshold values corresponding to layers in L is provided as another input. The threshold values serve as specific criteria for identifying similar neurons in terms of importance, and subsequently making neuron grouping decisions based on the similarity. The output of Algorithm 1 is set P of neuron partitions. Every partition $P_i = \{p^1, \dots, p^{ND}\} \in P$ determines how the neurons in layer $l_i \in L$ are distributed across the specified number of computing nodes ND.

The goal of Algorithm 1 is to evenly distribute the neurons n_j of every layer $l_i \in L$ over ND nodes (i.e., devices) in terms of importance. For example, applying Algorithm 1 (Lines 3–29) to the convolution layer with the five neurons n_1 to n_5 shown in Fig. 1 and setting $ND = 3$, the output P of the algorithm is the partition illustrated in Fig. 1(c). Algorithm 1 consists of three main steps performed on each layer $l_i \in L$.

In Step 1 (Lines 3–9), we first include each neuron $n_j \in l_i$ into a separate group G_j which is stored in the set of groups G (Line 5). Then, we calculate three importance scores for n_j from three different perspectives. The first score s_j^1 (Line 7) is the l_1-norm [36] which is a magnitude-based approach, widely used in CNN pruning techniques, to compute neuron importance based on the sum of its absolute weights and bias. The second importance score s_j^2 (Line 8) of n_j is computed by summing the sensitivity scores of all its connections with other neurons. We use the Taylor expansion approach [39] to obtain the connection sensitivity scores through the gradient in the propagation process [40]. The third score s_j^3 (Line 9) assesses the neuron importance by employing the Jensen–Shannon divergence [41] denoted as JSD. A larger change in the CNN output probability distributions $\mathbf{y}$, induced by removing neuron $n_j \in l_i$, indicates that n_j is more important. Instead of using a single importance score only, set $S_j = \{s_j^1, s_j^2, s_j^3\}$ of the three different scores enables a more comprehensive evaluation of the neuron importance because it performs a three-dimensional assessment of the importance, thereby facilitating a more effective clustering of neurons (see Table 1).

In Step 2 (Lines 10–20), Algorithm 1 takes the initial set of groups G created in Line 5, where each group contains only one neuron $n_j \in l_i$, and clusters these 1-neuron groups into a *new* set of groups G where any group may contain multiple neurons with similar importance. To this end, the following two actions are performed iteratively for every two groups $G_z \in G$ and $G_q \in G - G_z$. First, the largest distance d_{max} between the neurons in G_z and G_q is determined in Lines 12–17. Initially, d_{max} is set to zero. Then, for every pair of neurons $n_j \in G_z$ and $n_o \in G_q$, the Euclidean distance $d(n_j, n_o)$ between n_j and n_o in the three-dimensional importance score space (s^1, s^2, s^3) is computed in Line 15. If $d(n_j, n_o)$ is greater than d_{max}, then d_{max} is updated with $d(n_j, n_o)$ in Line 17.

Second, if d_{max} is below a given threshold value $t_i \in T$ then the neurons in G_z and G_q are merged (Line 19) into one group G_z because they are considered similar in terms of importance, and group G_q is removed from set G in Line 20. The threshold value t_i affects the result of the neurons clustering in Step 2. For example, a small t_i would result in set G having many groups with a few neurons per group. If t_i is too small then every group in G will contain only one neuron, thereby "forcing" the following Step 3 in Algorithm 1 to perform full replication of all neurons, thus maximizing the robustness at the expense of high resource requirements per node in the distributed system. In contrast, a large t_i would result in a few groups with many neurons per group. If t_i is too large then all neurons would be clustered into one group, thereby "forcing" Step 3 to perform very limited or no replication of neurons which could lead to a significant reduction of the robustness. Recall that a set T of threshold values t_i is given as an input to Algorithm 1, thus an optimal set of such values could be determined by integrating Algorithm 1 in a design space exploration (DSE) procedure with multiple

Algorithm 1: Robust Partitioning

Input : Set of layers L; Number of nodes ND;
Set of layer coefficients $W = \{W_1, ..., W_{|L|}\}$;
Set of threshold values $T = \{t_1, ..., t_{|L|}\}$;
Output: Set of neuron partitions $P = \{P_1, ..., P_{|L|}\}$;

1 $P \leftarrow \emptyset$
2 **for** $l_i \in L$ **do**
// Step 1: neuron importance scores
3 $G \leftarrow \emptyset$
4 **for** $n_j \in l_i$ **do**
5 Create G_j; $G_j \leftarrow G_j + n_j$; $G \leftarrow G + G_j$
6 Create $S_j = \{s_j^1, s_j^2, s_j^3\}$
7 $s_j^1 = \sum_{h=1}^{k_h} \sum_{w=1}^{k_w} \sum_{c=1}^{C_{in}} |W_j^{c,h,w}| + |b_j|$
8 $s_j^2 = \sum_{h=1}^{k_h} \sum_{w=1}^{k_w} \sum_{c=1}^{C_{in}} |\frac{\partial \mathbf{y}}{\partial W_j^{c,h,w}} \cdot W_j^{c,h,w}| + |\frac{\partial \mathbf{y}}{\partial b_j} \cdot b_j|$
9 $s_j^3 = \text{JSD}(\mathbf{y}_{\text{complete}} \,||\, \mathbf{y}_{\text{removing neuron } n_j})$
// Step 2: neuron clustering
10 **for** $G_z \in G$ **do**
11 **for** $G_q \in G - G_z$ **do**
12 $d_{max} = 0$
13 **for** $n_j \in G_z$ **do**
14 **for** $n_o \in G_q$ **do**
15 $d(n_j, n_o) = \sqrt{\sum_{a=1}^{3} (s_j^a - s_o^a)^2}$
16 **if** $d(n_j, n_o) > d_{max}$ **then**
17 $d_{max} = d(n_j, n_o)$
18 **if** $d_{max} < t_i$ **then**
19 $G_z \leftarrow G_z + G_q$
20 $G \leftarrow G - G_q$
// Step 3: round-robin distribution
21 Create $P_i = \{p^1, ..., p^{ND}\}$; $p^1 \leftarrow \emptyset, ..., p^{ND} \leftarrow \emptyset$
22 **for** $G_o \in G$ **do**
23 **if** $(|G_o| \mod ND) \neq 0$ **then**
24 **for** $j \in [1, ND - (|G_o| \mod ND)]$ **do**
25 Create $n_{|G_o|+j}$ = REPLICA($n_j \in G_o$)
26 $G_o \leftarrow G_o + n_{|G_o|+j}$
27 **for** $n_j \in G_o$ **do**
28 $r = (j \mod ND) + 1$; $p^r \leftarrow p^r + n_j$
29 $P \leftarrow P + P_i$
30 **return** P

Table 1
Top-1 accuracy (1D-Fail case in SysConf4D).

Importance Scores	AlexNet (%)	VGG16_BN (%)	ConvNext_Tiny (%)
s^1	43.718	60.426	76.618
s^2	43.642	58.920	75.904
s^3	43.432	59.942	76.134
$s^1 + s^2$	51.268	69.152	76.678
$s^1 + s^3$	51.658	71.736	76.580
$s^2 + s^3$	51.250	67.360	76.572
$s^1 + s^2 + s^3$	**52.396**	**72.500**	**76.820**

optimization objectives including distributed CNN inference accuracy, energy and resource requirements per node in the distributed system, and system performance.

Finally, in Step 3 (Lines 21–29), Algorithm 1 distributes all neurons n_j in every group $G_o \in G$ across a number of nodes ND in a round-robin fashion (Lines 27–28). If the number of neurons in group G_o is not a multiple of the number of nodes ND then some neurons in the group are replicated (Lines 23–26) in order to increase the neuron number to the closest multiple of the number of nodes before the round-robin distribution. Such round-robin distribution can guarantee that every node runs the same number of similarly important neurons from a group, thereby providing CNN model robustness by reducing the CNN inference accuracy degradation in the event of failures in the distributed system.

5. Evaluation of the RobustDiCE method

In this section, we present a range of experiments demonstrating the merits of RobustDiCE in terms of achieved robustness and resource utilization per node/device in a distributed system performing CNN inference.

5.1. Experimental setup

We implement RobustDiCE and apply it to the following distributed system configurations and real-world CNNs, and considering the following device failure scenarios.

CNNs and System Configurations: We experimented with three CNNs, namely AlexNet [42], VGG16-BN [2], and ConvNext-Tiny [43], taken from the TorchVision library. Given their extensive use in image classification and their diversity in layer types, operation counts, and memory requirements for weights, we consider these CNNs to be representative targets to demonstrate the merits of our method. By applying RobustDiCE, every CNN is distributed for inference on three system configurations: one with four edge devices (**SysConf4D**), one with three devices (**SysConf3D**), and one with two devices (**SysConf2D**). All devices in a system configuration are NVIDIA Jetson Xavier NX boards connected over a Gigabit network switch. Each device has an embedded MPSoC featuring a 6-core Carmel ARMv8.2 CPU, an NVIDIA Volta GPU with 384 CUDA cores, 48 Tensor cores, and 8 GB of LPDDR4x memory.

Device Failure Scenarios: For each of the aforementioned CNNs, we consider three scenarios.
Scenario A: The CNN is distributed for inference on system configuration **SysConf4D** where 1 device fails (**1D-Fail**), 2 devices fail (**2D-Fail**), or 3 devices fail (**3D-Fail**).
Scenario B: CNN on **SysConf3D** where **1D-Fail** or **2D-Fail**.
Scenario C: CNN on **SysConf2D** where **1D-Fail**.

Under every scenario with a different number of failing devices, we evaluate the preserved Top-1 accuracy on the ImageNet-1K test dataset when the CNN is distributed using our RobustDiCE method. We compare RobustDiCE to state-of-the-art robustness-unaware partitioning which performs filter and layer output partitioning, referred to as *LOP* [7], as well as the robustness-aware CDC method from [11]. In addition, we also show the Top-1 CNN accuracy results under an ideal scenario, called `Optimal`. This `Optimal` scenario assumes that in system configurations **SysConf4D**, **SysConf3D**, and **SysConf2D** no devices fail or all CNN neurons are replicated on every device in order to have quadruple (QMR), triple (TMR), and dual (DMR) modular redundancy, thus achieving maximum robustness.

By continuously providing 1000 images as an input data stream for the distributed CNN inference, we measure the system performance in images (frames) per second (FPS), memory usage per device in megabytes (MB), and energy consumption per device in joules per image (J/img) of the distributed CNN inference for the different system configurations. We measure the overall latency in processing the 1000 images and compute averaged FPS as throughput. The energy consumption per device is obtained with a special sampling thread that reads power values from the INA3221 power monitor, available on the NVIDIA Jetson Xavier NX board, where the power consumption involves the whole board including CPUs, GPU, SoC, etc. The sampled power values are integrated over the duration of the 1000-image inference, and the obtained energy consumption is divided by 1000 to represent it in J/img. The memory usage per device is reported directly by the executed CNN code itself during the CNN inference. Here, we strategically selected 1000 images to strike a balance between obtaining statistically accurate inference time metrics and maintaining a manageable experiment duration. Such a volume of input images is sufficient to fully engage the entire decentralized system and ensure that each node is fully utilized to reveal potential reliability issues under real operational loads. If the number of input images is too small, it will not put sufficient computation pressure on the system and result in inaccurate statistics of the system's operational capabilities. Conversely, too many input images will greatly extend the experimental duration without providing more information about the system's performance.

Timeout Configurations: To investigate the impact of the timeout mechanism (Section 4.1) on the overall latency per image of our robust distributed system against device failures, we conduct a comparison experiment using the VGG16-BN model deployed across four devices, i.e., deployed on the **SysConf4D** system configuration. We test the VGG16-BN overall latency under the **1D-Fail** scenario and compare it with the overall latency under a scenario without failures. In the scenario with failures, the timeout mechanism is applied utilizing different timeout values in the range of 10 to 30 000 µs. This comparison experiment reveals the overhead introduced by the timeout mechanism on the performance of our robust system.

5.2. Experimental results

Ablation Study of Importance Scores: To substantiate the efficacy of using multi-dimensional importance evaluation for neuron clustering, we carried out an ablation study with various combinations of importance scores (s^1, s^2, s^3). We evaluate the Top-1 accuracy of the 1D-Fail case for the SysConf4D system configuration in Table 1 using different combinations of importance scores. It is clear that the combination of all three scores preserves the Top-1 accuracy (model robustness) the best under the 1D-Fail scenario for all three models: 52.396% (AlexNet), 72.500% (VGG16_BN), and 76.820% (ConvNext_Tiny). These findings confirm the potential for enhancing model robustness in distributed CNN inference using the combination of multiple importance scores.

Impact of Timeout Mechanism: We present and analyze the impact of the timeout mechanism on the VGG16-BN end-to-end latency under scenarios with and without node failures utilizing the timeout configurations described in Section 5.1. The performance results are depicted in Fig. 3. The red horizontal, dashed line represents the measured overall latency per image in the scenario without failures where the timeout mechanism is not employed. For comparison, the blue curve depicts the VGG16-BN overall latency when failures occur during the inference process. In this scenario, the timeout mechanism

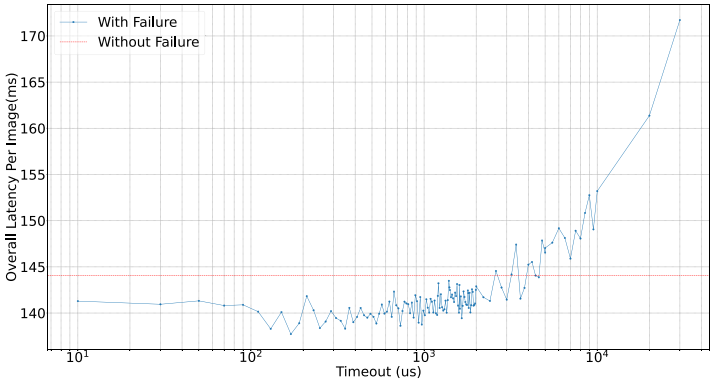
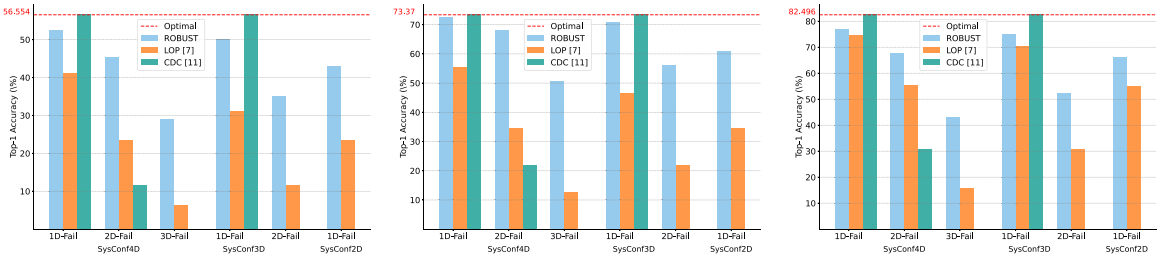


Fig. 3. Overall latency vs. Timeout against failures.



(a) AlexNet (b) VGG16-BN (c) ConvNext-Tiny

Fig. 4. CNN model robustness under different device failure scenarios.

is employed to ensure the continuation of the inference process despite node failures.

When the timeout value is set to be smaller (e.g., 500 μs) than the time needed to synchronize the results from multiple devices, the available devices ignore unfinished synchronizations between each other and proceed without waiting. This leads to a reduced overall latency, as shown in Fig. 3 where the latency of executions with a small timeout (blue line) can be lower than the latency for execution without failures/timeout (red line). However, setting a timeout value that is too small can also reduce the accuracy of the inference results, as it may not allow sufficient time for necessary communications to be completed.

Conversely, setting a relatively large timeout value (e.g., 30 ms) can cause the system to detect failures too late, leading to hanging communications until the timeout expires. This significantly increases the overall latency, rising up to 171.72 ms when failures occur, as the system waits for the extended timeout period before proceeding.

While the presence of failures and the subsequent application of the timeout mechanism do increase the VGG16-BN overall latency compared to a failure-free environment, the mechanism significantly mitigates the latency that would otherwise be caused by unresolved communication issues. This trade-off is essential for maintaining operational continuity and ensuring that the inference process can proceed even under less-than-ideal conditions.

Model Robustness Comparison: The results, obtained with the experimental setup described in Section 5.1, are presented in Fig. 4 and Table 2. For every CNN model, we show a graph where the X-axis represents the considered scenarios with a different number of failing devices, and the Y-axis indicates the evaluated Top-1 CNN model accuracy. For every scenario and number of failing devices, we plot a bar for the RobustDiCE results (blue bar), LOP results (orange bar), and CDC results (green bar). In addition, the horizontal dashed (red) line shows the accuracy under the `Optimal` scenario.

Looking at the blue and orange bars in Fig. 4, we observe that RobustDiCE consistently delivers higher Top-1 accuracy compared to the state-of-the-art but robustness-unaware LOP partitioning method. This clearly demonstrates the superiority of our method in terms of CNN model robustness. Taking Fig. 4(a) as an example, the Top-1 accuracy of AlexNet under the `Optimal` scenario is 56.55% which is our reference point. When a system configuration experiences device failures as in Scenario A, our RobustDiCE method delivers a Top-1 accuracy of 52.40%, 45.28%, and 28.93% for cases 1D-Fail, 2D-Fail, and 3D-Fail, respectively. In contrast, the LOP method exhibits more significant drop in accuracy, namely 41.07%, 23.50%, and 6.42% for the same device failure cases. A similar trend can be observed for VGG-16BN and ConvNext-Tiny in Fig. 4(b) and (c), respectively. Here, we have used an optimistic device failure scenario for LOP, i.e., devices with the least important groups of neurons fail.

Comparing our RobustDiCE method (orange bars) with the CDC method (green bars), we see that CDC is capable of perfectly handling a single device failure due to its approach of using actor replication and a spare node. However, the CDC method cannot handle multiple device failures, resulting in very low accuracy (much lower than RobustDiCE) or even complete failure (0% accuracy) when all but one devices fail.

Looking at Fig. 4 and comparing the Top-1 accuracy delivered by RobustDiCE with the reference accuracy under the `Optimal` scenario, we observe that our method does not maintain the reference accuracy level in the event of device failures. The reason is that, in this experiment, we set threshold values $t_i \in T$ discussed in Section 4 to be greater than 0. Because of this, our method does not replicate all CNN neurons on every device, thereby trading off CNN model robustness (loss of Top-1 accuracy) for reduced system resource utilization. This tradeoff could

Table 2
System performance and resource utilization.

Network	System Configuration	Max. per-device Energy (J/img)	System Throughput (FPS)	Max. per-device Memory (MB)
AlexNet	QMR/TMR/DMR	0.179	46.255	150.914
	CDC-SysConf3D	0.165	43.670	94.117
	CDC-SysConf4D	0.157	45.587	78.852
	Robust-SysConf2D	0.159	48.214	99.254
	Robust-SysConf3D	0.148	50.045	80.777
	Robust-SysConf4D	0.142	51.219	72.801
VGG16-BN	QMR/TMR/DMR	0.850	10.744	429.215
	CDC-SysConf3D	0.809	10.634	313.688
	CDC-SysConf4D	0.799	10.485	272.293
	Robust-SysConf2D	0.826	10.761	328.426
	Robust-SysConf3D	0.799	10.993	295.086
	Robust-SysConf4D	0.779	11.078	267.395
ConvNext-Tiny	QMR/TMR/DMR	0.308	28.223	88.895
	CDC-SysConf3D	0.307	27.107	69.129
	CDC-SysConf4D	0.297	28.248	59.961
	Robust-SysConf2D	0.301	28.044	76.465
	Robust-SysConf3D	0.296	28.415	65.203
	Robust-SysConf4D	0.288	29.034	58.090

be tuned by changing the t_i values. Moreover, if all t_i values are set to 0 then our method will maintain Top-1 accuracy at the same level as under the `Optimal` scenario. Under this scenario, all CNN neurons are replicated on every device in order to have quadruple (QMR), triple (TMR), or dual (DMR) modular redundancy, thus achieving maximum robustness. However, achieving this maximum robustness is at the expense of higher memory usage and energy consumption per device compared to the resource utilization, imposed by our method, when trading off robustness against utilization. This statement is supported by the resource utilization results in Table 2. In this table, for every CNN, we show the maximum per-device memory usage (Column 5), the maximum per device energy consumption (Column 3), and the overall system throughput (Column 4) for the three system configurations SysConf4D, SysConf3D, and SysConf2D with our RobustDiCE method and the CDC method as well as for the QMR/TMR/DMR configuration associated with the `Optimal` scenario.

System Performance: Considering the memory usage numbers for AlexNet, shown in Column 5, we see that the replication of all neurons on every device in system configuration QMR/TMR/DMR requires about 150 MB of memory per device. In contrast, our RobustDiCE method significantly reduces the required memory per device, i.e., with 51.76% for system configuration SysConf4D, with 46.47% for SysConf3D, and with 34.23% for SysConf2D. Significant memory reduction trends can be observed in Column 5 for VGG16-BN and ConvNext-Tiny as well. The memory usage numbers for CDC show that this method reduces the memory footprint in comparison to the all-neuron replication method (QMR/TMR/DMR) but still has higher memory usage compared to RobustDiCE.

The energy consumption per device is also reduced by RobustDiCE as compared to applying all-neuron replication to achieve CNN model robustness. For example, Column 3 in Table 2 shows that our method applied on SysConf4D achieves an effective energy reduction over the all-neuron replication method (QMR/TMR/DMR), i.e., 20.67% reduction for AlexNet, 8.35% for VGG16-BN, and 6.49% for ConvNext-Tiny. The CDC energy results again show an improved behavior compared to QMR/TMR/DMR but are inferior to the results from RobustDiCE.

Finally, as shown in Column 4 of Table 2, RobustDiCE slightly improves the system throughput for almost all CNNs and system configurations as compared to QMR/TMR/DMR (except for SysConf2D on ConvNext-Tiny). For CDC, on the other hand, the system throughput is generally lower than QMR/TMR/DMR and RobustDiCE.

We note, however, that the system throughput of distributed CNN inference is highly dependent on the quality of the network interconnecting the devices in the system. In our experiments, we have used a Gigabit network switch. Evidently, in other edge/IoT settings, the connectivity between devices may have a lower bandwidth, e.g., using WiFi or other wireless protocols. Thus, our RobustDiCE method cannot always guarantee system throughput improvements but it can guarantee memory usage and energy consumption reductions.

6. Discussion and conclusions

This paper presented our RobustDiCE approach for model and system robustness in distributed CNN inference at the Edge. Our implementation validates the system robustness of distributed CNN inference against possible device failures. By applying a robust partitioning method for distributing CNNs over multiple edge devices, our method preserves the model accuracy as much as possible against device/link failures. Several experiments demonstrated that RobustDiCE can retain the CNN model accuracy after failures much better as compared to the state-of-the-art partitioning methods. We have also shown the advantages of our RobustDiCE method over the optimal robustness approach and CDC method in terms of memory usage per device, energy consumption per device, and system throughput.

In terms of limitations of our decentralized framework for distributed CNN inference, scalability emerges as a primary challenge. Our decentralized framework, while robust in smaller settings, faces challenges in efficiently managing network traffic as the number of nodes increases. This limitation is crucial as it affects the framework's ability to scale up to large distributed systems without compromising performance.

Additionally, the security of distributed CNN inference is another concern that we have not addressed in our current work. The distributed nature of inference, requiring frequent communications between nodes, may introduce potential vulnerabilities. These vulnerabilities could expose the system to various attacks, undermining the reliability and trustworthiness of the distributed inference process.

In response to these issues, our future work will focus on developing more sophisticated mechanisms to enhance the scalability of our decentralized framework and studying the security aspects – also in relation to aspects such as robustness against failures – of distributed CNN inference. Specifically, we aim to implement advanced algorithms that can efficiently manage larger networks of nodes while minimizing communication overhead. Moreover, we plan to explore robust security methods that can safeguard the communication channels between nodes, thus enhancing the overall security posture of our distributed CNN inference system.

By addressing these critical areas, we hope to further improve the system's capability to operate reliably and securely on a larger scale, making it more suitable for widespread real-world applications.

CRediT authorship contribution statement

Xiaotian Guo: Investigation, Conceptualization, Methodology, Software, Experimental design and Implementation, Writing – original draft. **Quan Jiang:** Validation, Experimental test. **Andy D. Pimentel:** Supervision, Reviewing and editing. **Todor Stefanov:** Supervision, Reviewing and editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2009, pp. 248–255, http://dx.doi.org/10.1109/CVPR.2009.5206848.

[2] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2014, arXiv preprint arXiv:1409.1556.

[3] Z. Dai, H. Liu, Q.V. Le, M. Tan, Coatnet: Marrying convolution and attention for all data sizes, Adv. Neural Inf. Process. Syst. 34 (2021) 3965–3977.

[4] Y. Guo, A survey on methods and theories of quantized neural networks, 2018, arXiv preprint arXiv:1808.04752.

[5] T. Elsken, J.H. Metzen, F. Hutter, Neural architecture search: A survey, J. Mach. Learn. Res. 20 (1) (2019) 1997–2017.

[6] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, L. Tang, Neurosurgeon: Collaborative intelligence between the cloud and mobile edge, ACM SIGARCH Comput. Archit. News 45 (1) (2017) 615–629.

[7] R. Stahl, A. Hoffman, D. Mueller-Gritschneder, A. Gerstlauer, U. Schlichtmann, DeeperThings: Fully distributed CNN inference on resource-constrained edge devices, Int. J. Parallel Program. 49 (4) (2021) 600–624.

[8] X. Guo, A.D. Pimentel, T. Stefanov, Automated exploration and implementation of distributed cnn inference at the edge, IEEE Internet Things J. 10 (7) (2023) 5843–5858, http://dx.doi.org/10.1109/JIOT.2023.3237572.

[9] L. Zhou, M.H. Samavatian, A. Bacha, S. Majumdar, R. Teodorescu, Adaptive parallel execution of deep neural networks on heterogeneous edge devices, in: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, 2019, pp. 195–208.

[10] J. Mao, X. Chen, K.W. Nixon, C. Krieger, Y. Chen, Modnn: Local distributed mobile computing system for deep neural network, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, IEEE, 2017, pp. 1396–1401.

[11] R. Hadidi, J. Cao, B. Asgari, H. Kim, Creating robust deep neural networks with coded distributed computing for iot, in: 2023 IEEE International Conference on Edge Computing and Communications, EDGE, IEEE, 2023, pp. 126–132.

[12] E. Aghapour, D. Sapra, A. Pimentel, A. Pathania, CPU-GPU layer-switched low latency CNN inference, in: 2022 25th Euromicro Conference on Digital System Design, DSD, 2022, pp. 324–331, http://dx.doi.org/10.1109/DSD57027.2022.00051.

[13] Z. Zhao, et al., DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (11) (2018) 2348–2359, http://dx.doi.org/10.1109/TCAD.2018.2858384.

[14] R. Stahl, et al., Fully distributed deep learning inference on resource-constrained edge devices, in: International Conference on Embedded Computer Systems, Springer, 2019, pp. 77–90.

[15] R. Hadidi, J. Cao, M.S. Ryoo, H. Kim, Toward collaborative inferencing of deep neural networks on Internet-of-Things devices, IEEE Internet Things J. 7 (6) (2020) 4950–4960, http://dx.doi.org/10.1109/JIOT.2020.2972000.

[16] E. Tang, T. Stefanov, Low-memory and high-performance CNN inference on distributed systems at the edge, in: Proc. of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC, ACM, 2021, pp. 1–8.

[17] L. Zeng, X. Chen, Z. Zhou, L. Yang, J. Zhang, Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices, IEEE/ACM Trans. Netw. 29 (2) (2020) 595–608.

[18] X. Hou, Y. Guan, T. Han, N. Zhang, Distredge: Speeding up convolutional neural network inference on distributed edge devices, in: 2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2022, pp. 1097–1107.

[19] N. Laranjeiro, J. Agnelo, J. Bernardino, A systematic review on software robustness assessment, ACM Comput. Surv. 54 (4) (2021) 1–65.

[20] W. Cirne, F. Brasileiro, D. Paranhos, L.F.W. Góes, W. Voorsluys, On the efficacy, efficiency and emergent behavior of task replication in large distributed systems, Parallel Comput. 33 (3) (2007) 213–234.

[21] J.P. Walters, V. Chaudhary, Replication-based fault tolerance for MPI applications, IEEE Trans. Parallel Distrib. Syst. 20 (7) (2008) 997–1010.

[22] H. Tada, M. Imase, M. Murata, On the robustness of the soft state for task scheduling in large-scale distributed computing environment, in: 2008 International Multiconference on Computer Science and Information Technology, IEEE, 2008, pp. 475–480.

[23] S. Rajput, et al., DETOX: A redundancy-based framework for faster and more robust gradient aggregation, Adv. Neural Inf. Process. Syst. 32 (2019).

[24] N. Cheney, M. Schrimpf, G. Kreiman, On the robustness of convolutional neural networks to internal architecture and weight perturbations, 2017, arXiv preprint arXiv:1703.08245.

[25] K. Huang, P.H. Siegel, A. Jiang, Functional error correction for robust neural networks, IEEE J. Sel. Areas Inf. Theory 1 (1) (2020) 267–276.

[26] V. Amatya, A. Vishnu, C. Siegel, J. Daily, What does fault tolerant deep learning need from mpi? in: Proceedings of the 24th European MPI Users' Group Meeting, 2017, pp. 1–11.

[27] C. Liu, et al., Fault-tolerant deep learning: A hierarchical perspective, 2022, arXiv preprint arXiv:2204.01942.

[28] C. Torres-Huitzil, B. Girau, Fault and error tolerance in neural networks: A review, IEEE Access 5 (2017) 17322–17341.

[29] Z. Hakimi, Collaborative Inference for Distributed Camera System (Master's thesis), The Pennsylvania State University, 2019.

[30] S. Itahara, T. Nishio, K. Yamamoto, Packet-loss-tolerant split inference for delay-sensitive deep learning in lossy wireless networks, in: 2021 IEEE Global Communications Conference, GLOBECOM, IEEE, 2021, pp. 1–6.

[31] A. Yousefpour, et al., Resilinet: Failure-resilient inference in distributed neural networks, 2020, arXiv preprint arXiv:2002.07386.

[32] J. Boutellier, B. Tan, J. Nurmi, Fault-tolerant collaborative inference through the Edge-PRUNE framework, 2022, arXiv preprint arXiv:2206.08152.

[33] X. He, et al., AxTrain: Hardware-oriented neural network training for approximate inference, in: Proceedings of the International Symposium on Low Power Electronics and Design, 2018, pp. 1–6.

[34] A. Yousefpour, et al., Guardians of the deep fog: Failure-resilient DNN inference from edge to cloud, in: Workshop on Challenges in Artificial Intelligence and Machine Learning for IoT, 2019, pp. 25–31.

[35] J.L. Bernier, J. Ortega, E. Ros, I. Rojas, A. Prieto, A quantitative study of fault tolerance, noise immunity, and generalization ability of MLPs, Neural Comput. 12 (12) (2000) 2941–2964.

[36] H. Li, A. Kadav, I. Durdanovic, H. Samet, H.P. Graf, Pruning filters for efficient convnets, 2016, arXiv preprint arXiv:1608.08710.

[37] Y. He, G. Kang, X. Dong, Y. Fu, Y. Yang, Soft filter pruning for accelerating deep convolutional neural networks, 2018, arXiv preprint arXiv:1808.06866.

[38] Z. Hou, Y. Huang, S. Zheng, X. Dong, B. Wang, Design and implementation of heartbeat in multi-machine environment, in: 17th International Conference on Advanced Information Networking and Applications, 2003. AINA 2003., IEEE, 2003, pp. 583–586.

[39] N. Lee, T. Ajanthan, P.H. Torr, Snip: Single-shot network pruning based on connection sensitivity, 2018, arXiv preprint arXiv:1810.02340.

[40] S.-K. Yeom, P. Seegerer, S. Lapuschkin, A. Binder, S. Wiedemann, K.-R. Müller, W. Samek, Pruning by explaining: A novel criterion for deep neural network pruning, Pattern Recognit. 115 (2021) 107899.

[41] B. Fuglede, F. Topsoe, Jensen-Shannon divergence and Hilbert space embedding, in: Int. Symposium on Information Theory, 2004, p. 31.

[42] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, Commun. ACM 60 (6) (2017) 84–90.

[43] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, S. Xie, A convnet for the 2020s, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 11976–11986.