أكاديمية كاوست
KAUST ACADEMY

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Developing a Neural Network Aware Communication Framework for IoT devices

Raeid Fatah, Reena Aloraini, Dr. Clement Onime, Dr. Marco Zennaro

raeid.fatah@kaust.edu.sa, reena.aloraini@kaust.edu.sa, onime@ictp.it, Zennaro@ictp.it

Abstract

The growing adoption of Artificial Intelligence (AI) on edge devices has increased interest in distributing Neural Network (NN) computations across resource-constrained IoT devices. Recent literature has explored a variety of methods for distributed inference, with significant advances in optimizing memory consumption and reducing computational requirements. However, the communication required to coordinate inference across devices has received comparatively limited attention, while existing IoT communication protocols remain application-agnostic and leave neural-network-specific requirements, such as propagating intermediate outputs, synchronizing distributed computations, and handling execution dependencies, to the application layer. This work proposes NNAware, a high-level communication framework for mapping neural networks onto physically separated and resource-constrained IoT devices, enabling each device to act as an individual computing node during a forward inference pass. The framework provides mechanisms for device addressing, message formatting, input synchronization, transmission scheduling, transport abstraction, and lightweight failure handling. Experimental evaluation across five network topologies demonstrated correct distributed inference on physical IoT devices, with outputs matching independently simulated expected results across all tested configurations and recoverable failure scenarios. The results demonstrate the feasibility of incorporating neural-network-specific semantics into a reusable communication framework for distributed inference across IoT devices.

Date: August 24, 2026

Code: https://github.com/ReenaAloraini/NNAware

Contents

1 Introduction **2**

2 Related Work **2**

2.1 Network Organisation and Constraints . . . 2

2.2 Parallel Computing . . . 3

2.3 Communication and Synchronization . . . 3

2.4 Distributed Neural Networks Inference . . . 3

2.5 IoT Communication Protocols . . . 4

2.6 Research Gap . . . 5

3 Methodology **5**

3.1 System Overview . . . 5

3.2 Core Design Mechanisms . . . 6

3.3 NNAware Library . . . 7

3.4 Framework's Workflow . . . 9

3.5 Evaluation Procedure . . . 9

4 Results **11**

4.1 Experiments Overview . . . 11

4.2 Predictions Validation . . . 11

4.3 Execution Time . . . 12

4.4 Failover Behavior . . . 13

4.5 Unrecoverable Failure . . . 14

4.6 Unit Tests Results . . . 14

5 Discussion **14**

6 Limitations and Future Work **15**

7 Conclusion **15**

References **16**

1 Introduction

Neural Networks (NN) have become superior to other Machine Learning (ML) models due to their powerful capability of learning complex representations of data, leading to advancements in numerous fields such as computer vision, natural language processing, and healthcare. However, with these advancements, the complexity and computational demands of NN operations have significantly increased [1]. With the dramatic increase of the number of Internet of Things (IoT) devices over the recent years, distributed inference has emerged as a promising solution to the intensive computational requirements of NN and Deep Neural Networks (DNN), in which inference tasks are divided into subtasks that get distributed across resource-constrained edge and IoT devices [2]. Recent studies have improved the task of distributed inference through optimizing memory usage, computational efficiency, energy consumption, and latency, which allowed the deployment of large NN and DNN on resource-constrained devices while maintaining high accuracy [3][4][5][6]. However, where existing frameworks treat communication overhead between partitioned devices as a cost to be minimized [6][7], this work treats the communication layer as an object of study, as executing inference across multiple physical devices requires more than the simple transfer of data. Intermediate activations must be delivered to the correct successors, computation must be synchronized according to data dependencies, and failures must be detected without corrupting the inference process. In order to solve the problem of how independent, memory and power constrained IoT devices can be made to behave as a cluster to perform one forward inference pass, this work proposes a Neural-Network-Aware Communication framework for IoT devices. The framework allows users to map a pre-trained NN onto a group of physically-separated IoT devices, where each device is assigned a single NN node. The main contribution of this work is the introduction of NN semantics into the communication layer, providing reusable mechanisms for packet management, synchronization, transport abstraction, and lightweight fault tolerance for distributed forward inference.

2 Related Work

2.1 Network Organisation and Constraints

Currently, most frameworks utilize Neural Architecture Search (NAS) or other similar algorithms to distribute computations across IoT devices based on various constraints (e.g. devices that have higher memory and more ability to carry out complex calculations handle more computations than other less capable devices) [5][8][7][9]. On the other hand, other papers take the simpler route of compressing the whole network to make it fit the tiny constraints of the edge devices [10][11][12].

In the MCUNet paper [9], Ji Lin et al. discussed the issues of implementing deep neural networks on highly resource-constrained microcontrollers, where deep learning models exceed the memory limits even when quantized into int8. The MCUNet framework addresses these constraints through combining TinyNAS for designing the NN architecture, with TinyEngine for lightweight inference while optimizing memory and execution.

The paper by Lin and Wang [3] argued that although cloud-based processing and edge processing models in mesh networks might offer many advantages, they present a significant amount of latency. For that reason, the authors introduce the "Network as a Computer (NaC)" framework as an approach to in-network computation through mapping a neural network onto IoT devices, allowing the data to be processed as it travels through the network. Their work highlights the importance of network topology and node placement, as their framework operates under the

principle of "the locality of information processing", and mitigating the energy hole problem through avoiding the central gateway and better distributing the energy consumption of IoT devices.

2.2 Parallel Computing

With regards to parallelizing the computations in distributed neural networks, two methods are identified, Data parallelism, which simply splits the computations required between several devices (Applying the same task on multiple inputs) in other words, the same model is running on different inputs for each device. The other method has two structures, pipeline parallelism and model parallelism. Unlike data parallelism, these two structures divide the DNN's task into smaller subtasks, which are then divided by devices; this process supplements data parallelism. The difference between pipeline parallelism and model parallelism is simple with how these subtasks are divided, if the workload is divided sequentially, this is referred to as "pipeline parallelism". Whereas If it is divided in parallel, it is called "model parallelism". Model parallelism fully exploits the device's resources.[13][14]

2.3 Communication and Synchronization

There are three distinct architectures for dealing with communication and node synchronization.

1. centralised: Involves a centralised node (master node) which is responsible for coordinating the devices and co-inferencing. And worker nodes that assist the main node by carrying out a portion of the computations, these computations are aggregated at the end when they are sent back to the master node.
2. pipelined: Here, there isn't a single centralised node. Instead, everything is handled sequentially (each node receives input from the previous layer and sends its output to the next layer). Here, windows for computations and communication are typically determined beforehand.
3. decentralised: There isn't a single centralised node here. Each node must coordinate the data flow with every other node in the network, which increases communication overhead dramatically [13].

Alternatively, a different work by Yao et al. [7] introduces the idea of a staleness factor (α) , which is a parameter set by the user that determines the tolerable delay between computation and communication between nodes, where if $\alpha > 1$, nodes do not wait to receive all the data before starting computation. This is similar to the work of Lin and Wang [3], which introduces a framework where computation and communication are done at the same time.

2.4 Distributed Neural Networks Inference

To overcome the memory and computation constraints of implementing Deep Neural Networks on a single device, several studies discuss the concept of distributed neural networks at inference. A variety of distribution approaches have been implemented, where computations were divided by model, layer, operator, or task.[15][6][16].
The issues with cloud offloading include privacy concerns, high latency and communication costs [6], to address this, De Oliveira et al. [17] proposed DeepThings for distributing CNN inference across local IoT edge clusters.The framework include a Fused Tile Partitioning (FTP) to overcome the memory consumption of convolutional layers through dividing layers into independent tasks for each device. Moreover, the authors proposed a distributed work stealing method to allow the workload to be dynamic and balanced during inference through making idle devices steal

partitioned input data that are yet to be processed to speed inference. The framework also includes a reuse-aware work scheduling method to allow a more efficient work distribution through reducing redundant computations.
To address the same issue, Di Pascale et al. [3] introduced the concept of "Network as a Computer" (NaC) as part of their in-network computations framework, where neural networks are mapped onto IoT devices in mesh networks to allow for a simultaneous data aggregation and processing as data travel through the network, as opposed to sending raw sensor data directly to the cloud. The paper emphasised the cruciality of the placement of hidden nodes over available IoT devices, while considering transmission costs and network constraints. Thus, the author introduced a framework for the placement of hidden neurons that is subject to resource constraints by providing an energy-aware mapping of hidden neurons on a subset of IoT devices. The NaC concept showed improved energy consumption which extended the life time of devices, and a high reduction in information delivery latency, where average latency of distributed NN was half the latency of centralised (gateway) processing.
Alternatively, Teerapittayanon et al. [6] addressed the problem through proposing a Distributed Deep Neural Networks (DDNN) framework over three computing hierarchies; the cloud, edge devices (fog), and end devices distributed geographically. This approach works through the distribution of network depth; partitioning the neural network into layers, where the IoT devices handle shallow-layers such as the feature extraction and the classifier, then the edge layer and cloud handle more computationally intensive layers. The results illustrated in the paper show that DDNN allows for multiple end-devices to work in collaboration, reduces the communication costs by up to 20X, includes built-in fault tolerance with minimal effect on accuracy, and increases the data privacy within the system.
Similar to the work of Teerapittayanon et al. [6], the paper by Zhang et al. [4] introduced Heterogeneous Distributed Neural Networks (HDDNN) over the same three computing hierarchies (cloud, edge, and end devices). A key difference is that instead of splitting one model across the computing elements, their approach includes using different models for different devices based on the task and neural network structure, where small models such as MobileNet are implemented on IoT devices, and models requiring more computational capabilities such as ResNet are implemented in edge devices and the cloud. The findings showed that HDDNN optimized resource usage, had low response time, and high privacy protection.
Although collectively the work demonstrated in these papers show major advancements in approaches to distributed DNN on resource-constrained IoT devices in terms of partitioning, compression, and scheduling, they lack the focus on communication between the IoT devices, as studies in this area treat communication as a secondary supporting component for computation.

2.5 IoT Communication Protocols

In modern IoT systems, the Wireless Mesh Networks (WMN) have become fundamental due to abilities in improving coverage, their high reliability, and stronger fault tolerance compared to centralised architectures, as WMN allows communication to resume even when nodes or links fail, illustrating a self-organizing capability [18]. The survey by Cilfone et al. [18] compares the protocols suitable for mesh networks, and states that each IoT scenario requires a specific mesh protocol based on the application-specific requirements, as there is no superior mesh protocol that is optimal across all requirements.
In the context of communication protocols for distributed training and inference on IoT devices, Gulati et al. [19] proposed TDMiL (Tiny Distributed Machine Learning), a framework for distributed training and transfer learning on heterogeneous microcontroller-based IoT devices.

Because existing distributed learning and inference frameworks rely on communication stacks such as gRPC, TCP, and HTTPS, which are unsuitable for severely resource-constrained microcontrollers, the authors developed a lightweight communication stack based on the Constrained Application Protocol (CoAP) and Concise Binary Object Representation (CBOR). The framework introduces a central coordinator responsible for client registration, model distribution, parameter exchange, and coordination of the distributed learning process while supporting heterogeneous communication technologies. The results demonstrated the feasibility of distributed learning across multiple microcontroller platforms, as well as showing the impact of communication overhead and client coordination on performance. However, the framework is specifically designed for federated training, where complete or partial model parameters are exchanged between clients and a central server, rather than for coordinating distributed forward inference across physically separated neural-network nodes.

2.6 Research Gap

The existing literature showed significant evolution in implementing neural networks on resource-constrained IoT devices, with several methods of distributing inference across devices while optimizing memory and power consumption through model partitioning, compression, and resource-aware allocation of computations. However, these studies primarily focus on how computation is distributed, while the communication required to coordinate distributed execution is typically implemented as part of the proposed framework or application rather than as a reusable communication layer. Moreover, although current communication protocols for IoT devices provide a reliable packet delivery and network management required for distributed systems, they remain application-agnostic with respect to the execution of neural network, as they leave neural-network-specific tasks such as propagating activations, maintaining execution dependencies, synchronizing distributed computations, and handling node failures to the application itself. This becomes more significant for fine model partitions, where each physical device represents a single node, resulting in more frequent communication between the devices. Therefore, there is a need for a neural-network-aware communication framework that standardizes the exchange of inference information while remaining independent of both the core neural network architecture and the communication technology used.

3 Methodology

3.1 System Overview

The proposed system implements a neural-network-aware communication framework for distributed inference of neural networks across resource-constrained IoT devices. The communication framework defines how nodes, layers, and network clusters are addressed, how the computed outputs of nodes are transmitted throughout the network, and ensures reliability of delivery through coordinating execution times between nodes. A neural network is distributed across physically-separated IoT devices, where each device represents one node within the network that only stores its assigned weights, performs matrix multiplication and summation with the received input values from predecessor nodes, and applies the specified activation function to produce the computed value to be forwarded to nodes in the next layer. Figure 1 illustrates the system's architecture, and shows a key design contribution of separating the choice of communication protocol from the core framework through implementing an abstract interface for the transport operation, allowing the framework to operate independently of the reliability characteristics of a specific transport protocol.

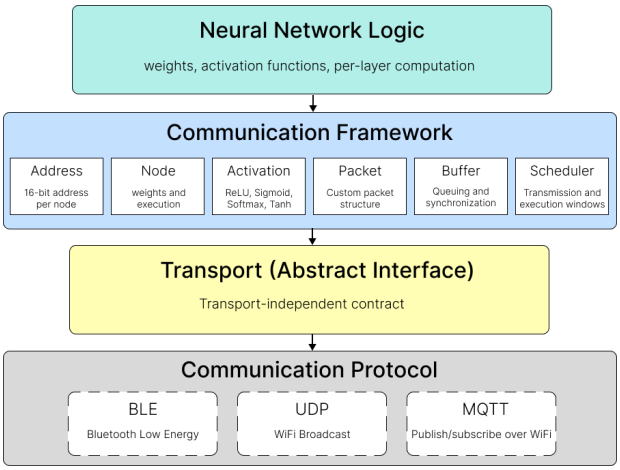


Figure 1 System architecture. The framework core depends only on the abstract Transport interface; the three transport implementations shown are interchangeable without modifying any component in the framework.

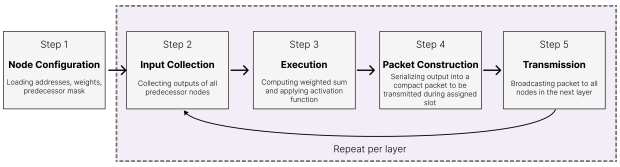


Figure 2 Operational flow of one inference pass through the network.

The system's operations at runtime are shown in Figure 2, a five-step per-layer cycle that begins with configuring nodes' information by assigning a unique 16-bit hierarchical address to each node, specifying the node identifier, layer identifier, cluster identifier, and a reserved 4-bit field for advanced configurations. Following that are five steps that are repeated at each layer until the network's final output is produced:

1. Input collection: the node receives the output of all nodes in the previous layer, tracked through a bitmask-gated buffer as a fault-tolerance mechanism to ensure that the input is complete.
2. Execution: the node computes a weighted sum of the received input and applies the activation function specified during configuration.
3. Packet Construction: the computed value is serialized into an application-layer packet, to be transmitted during the node's assigned transmission slot.
4. Transmission: the packet is broadcast to every node in the next layer through the active communication protocol implemented by the transport layer.

Reliability is ensured throughout the cycle by mechanisms such as packet integrity checking (checksum), time-out for failure detection, and retransmission of packets. Additionally, each node is configured with a backup role for one sibling node within the same layer, allowing it to compute and transmit on that node's behalf using a copy of its weights if it failed to transmit within the expected window, without requiring a dedicated backup device.

3.2 Core Design Mechanisms

Throughout runtime, three mechanisms account for most of the behavior of the framework at runtime through neural-network-awarness:

- **Buffer and Input Synchronization:** During input collection, each node maintains an input buffer where each predecessor node has a unique key that is stored alongside its sent value, and a bitmask is used to record which predecessors have sent their outputs, and the node repeatedly checks whether its input is complete before computing the weighted sum and activation function through comparing between the received mask and the expected predecessor mask that was assigned during setup. A bounded queue is used to examine incoming packets before storing them in the buffer, and drops duplicates.
- **Scheduler and Turn-Taking:** Each node observes the transmissions of its sibling nodes within a layer through the scheduler that tracks which nodes have transmitted against a configured turn-order mask that includes which sibling nodes must transmit before this node. This ensures an efficient and collision-free transmission based on observed traffic without requiring a synchronized clock.
- **Failover and Backup Weights:** During the setup phase, a node can act as a backup through optionally storing a copy of the weights, bias, and predecessor mask of one sibling node within the same layer, then during inference, this node constantly checks if all nodes except the target node have transmitted their output. If the target node failed to send its output, the backup node sends a resend request to the target node and waits a configured grace period for a reply. If no reply was sent, the target node is marked as a failed node, and the backup node takes its place through computing the output using the stored weights, and sends it under the failed node's address, flagged as a substitute value.

3.3 NNAware Library

The framework is implemented through custom C/C++ Arduino library designed to support the operations needed, consisting of the following modules in Table 1. In addition to the core communication modules, the library includes two supporting tools implemented in Python, a setup tool and a mapping tool, each with supporting modules shown in Table 2. The setup tool provisions IoT devices prior to inference by generating and transmitting the address and configuration data, to ensure that the devices' identities are known, and that the devices are ready to run inference. The mapping tool implements the logic for mapping a trained neural network model onto a set of IoT devices through parsing the model description, constructing the corresponding network topology, validating the mapping against per-device resource constraints, and generating the node configuration required for deployment, in addition to a simulation module to validate the mapping before deploying it in the hardware devices.
Each module was validated separately to ensure no dependency on the transmission protocol, to allow the substitution of the underlying protocol without modifying the core components of the framework.

Table 1 Library modules and their functionalities

Module	Functionality
`NNAddress.h`	16-bit node addressing (node / layer / cluster)
`NNPacket.h`	Application-layer packet format independent of any transport's native packet structure
`NNBuffer.h`	Input queuing and input synchronization for each node
`NNNode.h`	Stores weights, activation function, and execution logic
`NNScheduler.h`	Manages each node's transmission and execution timing, and transmission slots within a layer
`NNFailover.h`	Node-level fault tolerance: designated backup weights held per sibling node, event-driven failure detection, and retransmission requests
`NNTransport.h`	An abstract interface to separate the framework from the communication protocol

`NNTransportBLE`
`NNTransportUDP`
`NNTransportMQTT`
`NNTransportUDPMulticast.h`
	Implementations of `NNTransport`, each binding the framework to one specific communication protocol

Table 2 Python tools for devices setup and network mapping

Tool	Module	Functionality
Setup	`setup_tool.py`	Provisions devices with addresses and configuration prior to inference.
	`setup_messages.py`	Defines the setup-message format used during provisioning.
	`device_manifest.py`	Describes each device's identity and resource profile.
	`wire_format.py`	Serialization format shared between the setup tool and devices.
Mapper	`model_io.py`	Reads and parses trained neural network model descriptions.
	`topology.py`	Constructs the network topology for a given model and device set.
	`constraints.py`	Validates devices against resource constraints.
	`codegen.py`	Generates per-device configuration and code for the mapped network.
	`simulate.py`	Simulates a mapping prior to hardware deployment.

3.4 Framework's Workflow

The framework (locally hosted) is designed to allow users to deploy and run a pre-trained Neural Network while providing flexible options and clear instructions. The workflow of the framework follows a fixed sequence of steps:

- **Model Description:** a pre-trained model is uploaded or written directly in the framework in simple JSON description.
- **Model Mapping and Validation:** the framework expands the model into a per-node topology, informs the user of the required number of physical devices, and uses a Python simulation to compute the expected output of the network.
- **Optional Assignment of Backup Roles:** as described in Section 3.2, a user can select which nodes can act as a backup in case of a failure of a sibling node within the same layer. The tool validates each pairing and flags any pairing whose recovery will be structurally slower.
- **Scanning (device discovery):** unconfigured devices broadcast a discovery "HELLO" message containing their hardware ID, the framework listens to the devices over a fixed time window configured by the user and lists all available devices for the user to choose from.
- **Setup (provisioning):** once the desired devices are selected, the setup tool provisions each one in sequence and verifies the setup with an assemble checksum before confirming the configuration of the device, then broadcasts a "START" message to the devices.
- **Running:** the user provides the input vector and starts the inference, and each device repeats the per-layer inference operations shown in Section 3.1 Figure 2 until the final output is produced, alongside the output value of each node compared to the expected values computed during the mapping phase.
- **Failure handling:** the user is informed if a backup needed to be used due to the failure of a device, and which devices failed. In case a device was not backed up or recovery was not possible within the configured execution window, the user receives a timeout message with a description of the non-responsive devices.

3.5 Evaluation Procedure

3.5.1 Tools, Libraries, and Frameworks

The framework was developed and evaluated using a combination of tools summarized in Table 3. Each physical node runs on a Seeed Studio Wio Terminal, which includes a module that provides support for 2.4G/5G WiFi and Bluetooth Low Energy 5.0. The choice of this board was mainly to represent the memory and power constraints of the IoT devices that this framework is designed for, and the integrated WiFi and Bluetooth radios allowed for many transport protocols to be applied and tested without using additional peripherals.

Table 3 Development tools and libraries used in the framework

Category	Tool / Library
IoT devices	Seeed Studio Wio Terminal (ATSAMD51 Cortex-M4F, 120 MHz, 192 KB RAM, 4 MB flash, RTL8720DN WiFi + BLE 5.0)
Development environment	Arduino IDE, VS Code, Seeed SAMD Boards package (v1.8.6)
BLE transport	Seeed Arduino `rpcBLE`
WiFi transport	Seeed Arduino `rpcWiFi`
MQTT client	`PubSubClient`
MQTT broker	Eclipse Mosquitto
Independent verification	nRF Connect
C++ Desktop Testing	GCC/g++, CMake
Python testing	pytest
Model description and parsing	JSON
Framework UI	streamlit (v1.60.0)

Although transport implementations were developed for multiple communication protocols, hardware evaluation in this work was limited to UDP broadcast and UDP multicast. BLE could not be evaluated due to limitations encountered with the available BLE library for the selected hardware, as discussed further in Section 6.

3.5.2 Test Input

Test inputs were created manually to ensure consistency with the developed system, as the aim of testing is to ensure the functionality of communication and coordination within the framework. The following specific test inputs were used:

- Test vectors: synthetically created input vectors consisting of weights and input values for each node to ensure that nodes can compute the expected value and apply the selected activation function correctly.
- Network topologies: several synthetic network topologies were constructed to test specific functionalities of the system, for example, a network of two devices was created to test the transport abilities of devices using different communication protocols.
- Packet payload test: to ensure that the packets' content remains unchanged after serialization and deserialization, manually selected known byte sequences were transmitted and verified.

3.5.3 Python Tests

To validate the protocol layer independently of the hardware, a set of Python test scripts were created to test the data transmission and reception through an implementation of the UDP transport, alongside a set of pytest-based test to verify packet-level correctness and cross-language compatibility between C++ and Python, without requiring hardware devices. Over 65 tests were performed under the following categories:

- Packet validation: verifying that correctly-formed packets of each defined type (DATA/-

CONTROL/ACK) are accepted, and that invalid packets (corrupted checksums, truncated or oversized payloads) are rejected.

- Fault injection: simulating unreliable transmission conditions through corrupting, delaying, duplicating, reordering, and dropping packets
- Cross-language compatibility: ensuring that packets produced by the Python implementation are interpreted identically by the C++ implementation and vice versa, verifying that all fields, byte content, and checksums remain consistent throughout.

These tests establish that the packet format is implemented consistently across both languages and is robust to the fault conditions expected in a real deployment.

4 Results

4.1 Experiments Overview

To test the framework, five different Neural Network topologies differing in depth were designed and tested on Wio Terminal boards, scaling from a single-device AND-gate topology and up to a five-device topology with three layers. The input layer (Layer 0) is virtual by default in the framework and across all used topologies, this saves one physical device per input feature. However, the framework supports the use of physical input devices. All experiments used the fixed runtime settings summarized in Table 4.

Table 4 Fixed runtime parameters

Parameter	Value
Setup: device discovery window	10.0 s
Setup: retries per message	5
Setup: wait per attempt	0.5 s
Running: timeout waiting for a result	10.0 s
Failover resend grace	300 ms

4.2 Predictions Validation

Table 5 summarizes all five conducted experiments across five topologies. In every trial, the distributed inference output matched the expected output generated by the simulated python run prior to running each topology on the physical devices. The deepest topology tested was in Experiment 5, the output of each node is shown in Table 6, which confirms that correctness holds across every node in the forward pass and not only at the output node.

Table 5 Prediction vs. expected output by experiment

Exp.	Devices	Topology Shape	Input	Activation	Prediction	Expected
1	1	2 → 1	(0, 0)	ReLU	0.0	0.0
2	2	2 → 1 → 1	(1, 0.5)	ReLU, Sigmoid	0.613	0.613
3	3	2 → 2 → 1	(0.5, 1.5)	ReLU, Sigmoid	0.3252	0.3252
4	4	3 → 3 → 1	(1, -0.5, 0.5)	ReLU, Sigmoid	0.5793	0.5793
5	5	2 → 2 → 2 → 1	(2, -1.0)	ReLU, tanH, Sigmoid	0.6741	0.6741

Table 6 Experiment 5 per-node validation results

Node	Layer	Physical Output	Expected
`L1_N0`	1	1.300000	1.300000
`L1_N1`	1	0.000000	0.000000
`L2_N0`	2	0.744277	0.744277
`L2_N1`	2	-0.345214	-0.345214
`L3_N0`	3	0.674142	0.674142

4.3 Execution Time

The execution time of the distributed inference pass of each topology is shown in Figure 3. Execution time was initially expected to increase as the number of nodes increases, however, the execution time of the four-device topology (Experiment 4) with one hidden layer is higher (414.5 ms) than the execution time of the five-device topology with two hidden layers (328.0 ms). This latency difference can be explained by the difference of the node distribution between the two topologies, deeper five-device topology contains two hidden layers with two nodes each, whereas the four-devices topologies contains one hidden layer with three nodes, suggesting that layer width and the resulting communication and synchronization requirements may significantly influence latency more than the network depth or device count.

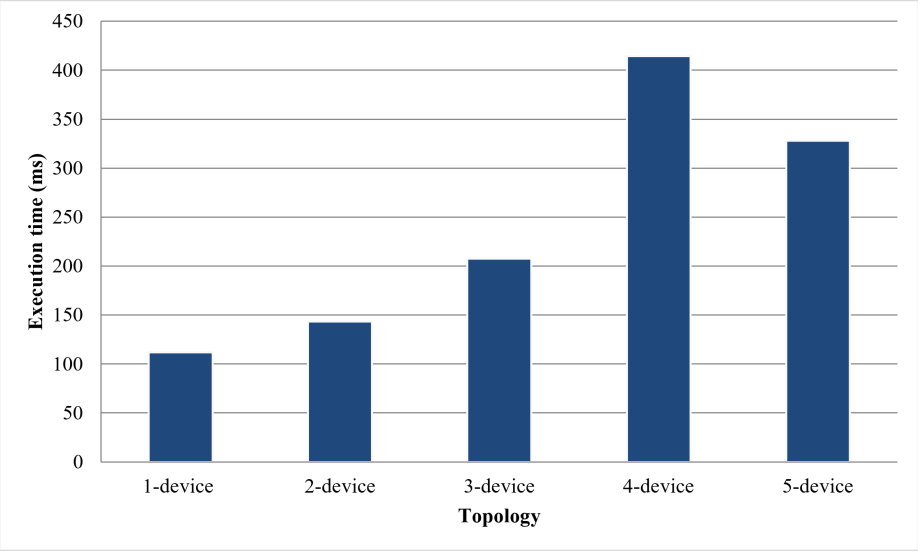


Figure 3 Execution time by topology

4.4 Failover Behavior

To test the failover behavior of the framework, four failover scenarios were tested, in which one or more devices were turned off during the inference. In Experiment 3, one of the two nodes within the hidden layer was turned off, and the other node acted as a backup as set during the configuration phase. In Experiment 4, failure occurred in the three-node layer where the failed device was assigned to the middle node, with a sibling node waiting behind it, resulting in higher overhead. Experiments 5a and 5b were conducted on the five-device topology, where in Experiment 5a one device failed, and in 5b two devices failed simultaneously in two different layers, both in the last-transmit position. Table 7 and Figure 4 summarize the results; every test case produced the correct failover behavior from the assigned backup nodes, and the correct inference results after substitutions.

Table 7 Execution time before and after failover

Exp.	Failed Nodes	Backup Nodes	Baseline (ms)	After Failover (ms)	Overhead (ms)
3	`L1_N0`	`L1_N1`	207.5	511.3	+303.8
4	`L1_N1`	`L1_N0`	414.5	2167.1	+1752.6
5a	`L1_N1`	`L1_N0`	328.0	303.5	–24.5
5b	`L1_N1, L2_N1`	`L1_N0, L2_N0`	328.0	840.0	+512.0

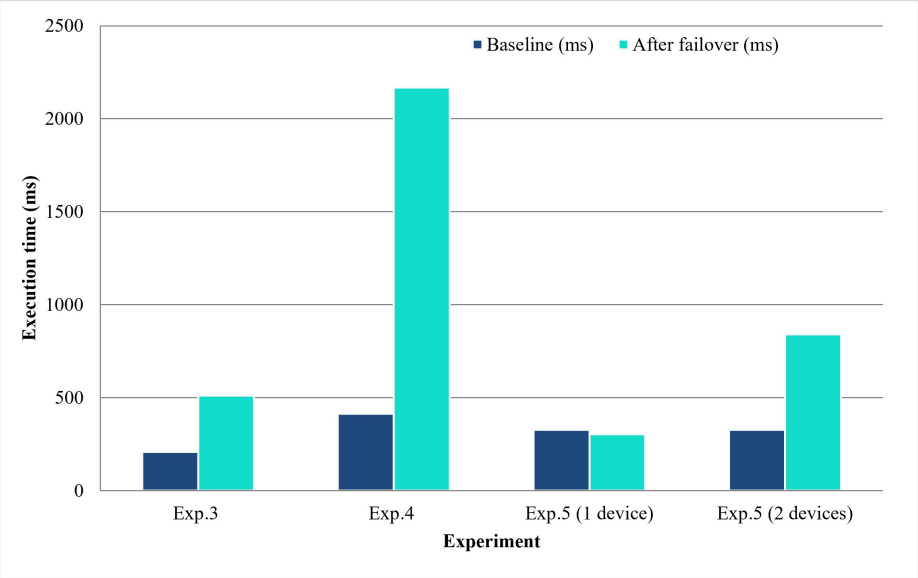


Figure 4 Execution time before vs. after failover

4.5 Unrecoverable Failure

In Experiment 3, a failure scenario involved killing the only node in the last layer which has no backup, producing an inference timeout after the configured 10.0 seconds inference window, rather than an incorrect or fabricated output. This timeout result was the correct and expected result, as the framework has no way to recover from the loss of an unprotected node, and it fails visibly rather than silently.

4.6 Unit Tests Results

Independently of the hardware trials above, the Python test suite described in Section 3.5.3 produced 65/65 passing results across its three categories: packet validation, fault injection, and cross-language (C++/Python) compatibility. These tests establish that the packet format and its handling of corrupted, delayed, duplicated, reordered, and dropped packets is consistent and robust, independent of the specific hardware runs reported above.

5 Discussion

Correctness holds across network scale and failures: The results showed that across each and every topology tested, and across all failure scenarios, the final output of the distributed inference on physical devices matched the output of the Python simulation, most importantly demonstrating that accuracy was maintained across tested topologies and failure conditions, including node substitution for failure handling. This also supports the concept this framework is designed for: a group of physically-separated and resource-constrained devices can be made to collectively behave as a network performing a single correct forward inference pass.

Execution time depends on topology structure and layer width: As shown in Section 4.3, although the topology in Experiment 4 used fewer devices than the topology in Experiment 5, its execution time was slower, and this was likely due to the difference in the distribution of the nodes within a layer. Experiment 5 is a deep three-layer five-device topology with two nodes in each hidden layer, executed in 328.0 ms, whereas Experiment 4 is a shallower two-layer four-device topology with three nodes in the hidden layer, executed in 414.5 ms. This can be explained by the framework's serialized communication within a layer, and as the width of a layer increases, more

nodes take transmission turns, and the successor nodes in the next layer cannot execute until each one receives all inputs from the previous layer. Therefore, wider layers in a network may increase latency by adding communication and synchronization overhead more than a deep narrower network structure.

Transmission order of nodes affects failover latency: Throughout the four failover cases, a unique pattern was observed based on the position of the failed device and the resulted latency: the recovery overhead was strongly influenced by the failed node's position within a layer, because the nodes' positions determine the transmission order. In Experiment 5a, the failed node was the last node in the layer to transmit, meaning no sibling nodes were waiting for it to transmit, resulting in an execution time of 303.5 ms compared with the 328.0 ms baseline execution time of the topology, meaning the recovery time did not add any overhead beyond the normal variance in time between runs. In contrast, in Experiment 4, the failure occurred in a node in the middle position within the layer, meaning another node was waiting for its transmission turn, resulting in a substantially larger overhead with 1752.6 ms increase in execution time. This recovery time overhead can be explained by factors in the timing mechanisms within the framework such as the default resend-grace period of 300 ms, and the 1500 ms transmission-turn stall timeout for nodes blocked behind a failed node. Additionally, in Experiment 5b, failure occurred in two devices in two separate layers, and because both nodes were in the last transmission position, the resulting overhead was 512.0 ms only, showing that multiple failures within a forward pass do not necessarily produce large latency as long as the transmission turn of other nodes was not blocked.

6 Limitations and Future Work

During the development of this work, a few limitations were encountered. Firstly, the limited number of hardware devices available restricted the scalability testing of the framework and prevented evaluating it with more realistic neural network topologies. Secondly, the currently available version of the Seeed's BLE library for Wio Terminal devices had limited documentation and internal bugs within the BLE Advertiser and Scanner modules, which restricted the testing of BLE as a communication protocol in the framework. Moreover, the current failover mechanism has no support for single-node layers, since backup roles can only be assigned between sibling nodes in the same layer. Therefore, future work should improve the failover mechanism to be suited for layers containing one node to avoid terminating execution of the network in case of a failure in that layer. Testing other communication protocols would provide stronger grounds for the framework's independence from the underlying transport technology used, as the current implementation only tested UDP broadcast and UDP multicast. Finally, applying a resource-aware workload distribution would be a powerful extension to this work, allowing devices with higher compute power to take on more computations, which can make mapping large and complex deep learning networks across IoT devices more efficient.

7 Conclusion

This work proposed and implemented a Neural-Network-Aware Communication framework for distributing neural network inference across physically separated and resource-constrained IoT devices. The NNAware framework addresses the communication requirements of distributed inference through providing neural-network-specific mechanisms for addressing, packet management, input synchronization, transmission scheduling, failure handling, and abstraction of the underlying communication protocol, enabling physically separated devices to collectively

perform a single forward inference pass. The experiments conducted showed that the framework was able to correctly execute different neural network topologies across multiple physical devices, with the distributed outputs matching the expected simulated results in all tested cases, including recoverable failure scenarios where backup nodes had to substitute the failed nodes. The results also showed that the structure of the mapped topology and the transmission order of nodes can affect execution and failover latency, emphasizing the importance of communication-aware network mapping. Overall, the developed framework demonstrates that incorporating neural-network-specific semantics into communication can provide a reusable approach for coordinating distributed inference across IoT devices, establishing a foundation that can be extended to larger networks, additional communication technologies, and more advanced workload distribution strategies.

References

[1] M. Threadgill and A. Gerstlauer. "A Survey of Distributed Learning in Cloud, Mobile, and Edge Settings". In: *arXiv preprint arXiv:2405.15079* (2024). DOI: 10.48550/arXiv.2405.15079. arXiv: 2405.15079. URL: https://arxiv.org/abs/2405.15079.

[2] Y. Yan et al. "DIDS: A Distributed Inference Framework with Dynamic Scheduling Capability". In: *Future Generation Computer Systems* 162 (July 2024), p. 107459. DOI: 10.1016/j.future.2024.07.037.

[3] E. Di Pascale et al. "The Network as a Computer: A Framework for Distributed Computing over IoT Mesh Networks". In: *IEEE Internet of Things Journal* 5.3 (June 2018), pp. 2107–2119. DOI: 10.1109/JIOT.2018.2823978.

[4] Z. Zhang et al. "Towards Ubiquitous Intelligent Computing: Heterogeneous Distributed Deep Neural Networks". In: *IEEE Transactions on Big Data* 8.3 (2022), pp. 644–657. DOI: 10.1109/TBDATA.2018.2880978.

[5] B. Huang, A. Abtahi, and A. Aminifar. "Energy-Aware Integrated Neural Architecture Search and Partitioning for Distributed Internet of Things (IoT)". In: *IEEE Transactions on Circuits and Systems for Artificial Intelligence* 1.2 (Nov. 2024), pp. 257–271. DOI: 10.1109/TCASAI.2024.3493036.

[6] S. Teerapittayanon, B. McDanel, and H. T. Kung. "Distributed Deep Neural Networks over the Cloud, the Edge and End Devices". In: *arXiv preprint arXiv:1709.01921* (2017). DOI: 10.48550/arXiv.1709.01921. arXiv: 1709.01921.

[7] J.-L. Lin and S.-D. Wang. "Communication-Efficient Separable Neural Network for Distributed Inference on Edge Devices". In: *arXiv preprint arXiv:2111.02489* (2021). arXiv: 2111.02489. URL: https://arxiv.org/abs/2111.02489.

[8] F. M. C. De Oliveira et al. "PANCODE: Multilevel Partitioning of Neural Networks for Constrained Internet-of-Things Devices". In: *IEEE Access* 11 (Jan. 2023), pp. 2058–2077. DOI: 10.1109/ACCESS.2023.3234245.

[9] Ji Lin et al. "MCUNet: Tiny Deep Learning on IoT Devices". In: *arXiv preprint arXiv:2007.10319* (2020). arXiv: 2007.10319. URL: https://arxiv.org/abs/2007.10319.

[10] P.-E. Novac et al. "Quantization and Deployment of Deep Neural Networks on Microcontrollers". In: *Sensors* 21.9 (Apr. 2021), p. 2984. DOI: 10.3390/s21092984.

[11] K. Bhardwaj et al. “Memory- and Communication-Aware Model Compression for Distributed Deep Learning Inference on IoT”. In: *arXiv preprint arXiv:1907.11804* (2019). arXiv: 1907.11804. URL: https://arxiv.org/abs/1907.11804.

[12] Shaoxiong Yao et al. “DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework”. In: *arXiv preprint arXiv:1706.01215* (2017). DOI: 10.48550/arXiv.1706.01215. arXiv: 1706.01215.

[13] T. Elgamal and K. Nahrstedt. “Serdab: An IoT Framework for Partitioning Neural Networks Computation across Multiple Enclaves”. In: *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2020, pp. 519–528. DOI: 10.1109/CCGRID49817.2020.00041.

[14] N. Solé and J. Miquel. “Horizontally Distributed Inference of Deep Neural Networks over Resource-Constrained IoT Networks”. In: *IEEE Internet of Things Journal* 10.5 (2023), pp. 3995–4005. DOI: 10.1109/JIOT.2022.3157552.

[15] F. N. Peccia and O. Bringmann. “Embedded Distributed Inference of Deep Neural Networks: A Systematic Review”. In: *arXiv preprint arXiv:2405.03360* (2024). arXiv: 2405.03360. URL: https://arxiv.org/abs/2405.03360.

[16] I. Rodriguez-Conde, C. Campos, and F. Fdez-Riverola. “Horizontally Distributed Inference of Deep Neural Networks for AI-Enabled IoT”. In: *Sensors* 23.4 (2023), p. 1911. DOI: 10.3390/s23041911.

[17] Z. Zhao, K. M. Barijough, and A. Gerstlauer. “DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2348–2359. DOI: 10.1109/TCAD.2018.2858384.

[18] A. Cilfone et al. “Wireless Mesh Networking: An IoT-Oriented Perspective Survey on Relevant Technologies”. In: *Future Internet* 11.4 (Apr. 2019), p. 99. DOI: 10.3390/FI11040099.

[19] Mayank Gulati et al. “TDMiL: Tiny Distributed Machine Learning for Microcontroller-Based Interconnected Devices”. In: *IEEE Access* 12 (2024), pp. 167810–167826. DOI: 10.1109/ACCESS.2024.3492921.