# Experimental test-bed for Computation Offloading for Cooperative Inference on Edge Devices

Nicholas Bovee
bovee@rowan.edu
Rowan University
Glassboro, New Jersey, USA

Stephen Piccolo
piccol24@students.rowan.edu
Rowan University
Glassboro, New Jersey, USA

Shen-Shyang Ho
hos@rowan.edu
Rowan University
Glassboro, New Jersey, USA

Ning Wang
wangn@rowan.edu
Rowan University
Glassboro, New Jersey, USA

## ABSTRACT

In this paper, we describe the experimental test-bed we are developing to evaluate the efficacy of computation offloading for cooperative inference without in-depth architectural changes to the models under consideration. We describe our test-bed design, functionalities, and work in progress features. We demonstrate a simple use case of the test-bed, consisting of the identification of a split layer in an AlexNet for a particular hardware scenario and the validation of those results.

## CCS CONCEPTS

• **Computing methodologies** → **Machine learning approaches**.

## KEYWORDS

deep learning, edge computing, split computing, cooperative systems

**ACM Reference Format:**

Nicholas Bovee, Stephen Piccolo, Shen-Shyang Ho, and Ning Wang. 2023. Experimental test-bed for Computation Offloading for Cooperative Inference on Edge Devices. In *The Eighth ACM/IEEE Symposium on Edge Computing (SEC '23), December 6–9, 2023, Wilmington, DE, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3583740.3626814

## 1 INTRODUCTION

Computation offloading is the "task of sending computation intensive application components to a remote server" [1]. Computation offloading from edge devices to cloud ensures that energy is not excessively consumed and more powerful processors (on the cloud server) are used in the computation. This is especially important when one is performing intensive inference tasks using complex prediction models (e.g., neural network) with a deep architecture. The main challenge in computation offloading is the communication latency between an edge device and the cloud server.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SEC '23, December 6–9, 2023, Wilmington, DE, USA*
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0123-8/23/12…$15.00
https://doi.org/10.1145/3583740.3626814

The experimental test-bed described in this paper takes hints from similar efforts such as "AIoTBench" by Luo et. al [9], and aims to facilitate the evaluation of computation offloading approaches for deep neural network (DNN) models for cooperative inference without underlying custom implementations of the models in evaluation. It is our objective to be able to apply computational offloading to pretrained models and to models normally not executable on edge devices due to resource (e.g., power, storage, etc) constraints by the use of this test-bed.

In Section 2, we describe computational offloading strategies for cooperative inference. Section 3 describes in detail the test-bed design, functionalities, and work in progress. In Section 4, we demonstrate a simple use case of the test-bed. Finally, we describe extension to the test-bed and future work in Section 5.

## 2 COMPUTATIONAL OFFLOADING FOR COOPERATIVE INFERENCE

One main step in the offloading process is the division of the task into offloadable and non-offloadable partitions such that the non-offloadable part is on an edge device and the offloadable is on the cloud server. With a deep learning prediction model the intuitive approach is to divide the inference computation of the deep learning model architecture at a layer which is easy to transfer to a cloud server. Of the total layer sequence, $k$, with $n$ layers and split point $j_c$, the early portion of the DNN, layers $[1, j_c)$ will be computed on the edge device. The later portion, layers $[j_c, n]$ will be computed at the cloud server, typically including the final prediction mapping. For such an offloading process, the inference accuracy (compared to inference without computation offloading) is minimally affected by changes in the environment, but there is no loss of information due to encoding or compression.

Kang et. al[7] first proposed "Neurosurgeon" that identifies the split in DNN model to minimize total latency or energy consumption. Recent development of such offloading approach can be found in [12]. A subdivision of these splits is the application of artificial bottlenecks into models that do not present natural bottlenecks[11], such as ResNet [5], though encoders or quantization[4][2]. In general, the "offloaded" portion of the model does not need to be on a true cloud server; any device with greater processing power could serve the same purpose.

Beside layer-wise splitting of model, one can consider input splitting (with overlaps)[13]. Another consideration in terms of the

strategy employed for cooperative inference is the use of distributed computation[10]. More discussions of offloading approaches can be found in[3][6] (and herein).

There are two factors that one can optimize to maximize the efficiency of the inference task when offloading is performed, namely: communication latency (due to data transfer between an edge device and the server) and model computation time (due to limitation of processing speed on edge devices). One selects the split point so that the combined communication latency and the computation time for the inference task is minimized.

Communication latency depends on: (1) size of output data from the split point, and (2) network bandwidth. In deployed models, it is difficult to control the network bandwidth available to the edge device and data transfer to the cloud server, and thus many implementations are focused on the reduction of data to be transferred. Computation time depends on the processing hardware on the edge device and cloud server, and the characteristics of the model in use.

# 3 TEST-BED DESIGN AND FUNCTIONALITIES

## 3.1 Design Overview

The test-bed is designed to provide a high-level interface that separates the test designer from setup and maintenance tasks that make it difficult to explore new techniques without manually reconfiguring each device's environment, the local network, and the benchmarking result collection process. The result is a system that allows users to quickly define experiments in terms of which device assumes which user-defined role, and which sequence of required tasks is assigned to each device. Network configuration, runtime environments, and benchmark collection are all handled by the test-bed.

To accomplish this, the test-bed API interprets a user-provided YAML (Yet Another Markup Language) file, serving as a manifest for the experiment. Each custom service is distributed to a temporary environment on each specified device to deploy the *Participating Nodes*. Concurrently, an *Observer Node* is deployed on the local machine to await the complete transfer of the test setup, delegate tasks to the Participating Nodes, and finally send a start signal to begin the experiment. During the test, intermediate tensors, parsed results, and gradients are shared through RPC (Remote Procedure Call) as required by the Participating Nodes. The Observer Node monitors the Participating Nodes through RPC, and centrally gathers the test results for display, interpretation, and further data process. Figure 1 provides a block diagram of how the various elements of the test-bed interact.

Our current version of the test-bed aims to provide a testing environment in which models can be evaluated for use or deployment without any code level modification to the model itself. This flexibility allows for rapid development, in that custom sub-classes of a particular model are not needed to begin testing candidacy in edge computing. The current implementations support usage of models without architecture modification, and with additional development will be able to support some varieties of bottleneck injection.

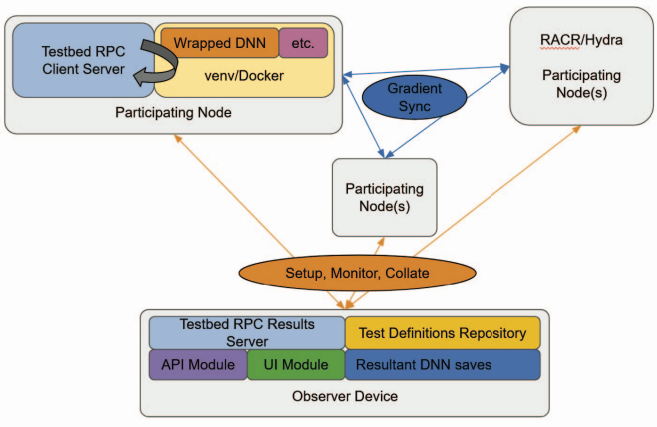


**Figure 1: Test-bed Design Overview**

## 3.2 Observer Device

The core implementation of the test-bed runs on the Observer Device. It is split into two distinct sub-packages: the `experiment_design` package, providing base classes and helper functions designed to be easily extend into custom test case components, and the `app_api` package, which powers the test-bed itself and is not designed to be adjusted by the user.

*3.2.1 Experiment Design Package.* This package is split into subsections corresponding to each basic component that defines the way a node behaves during testing:

- *Datasets*: How the node receives input data for processing
- *Models*: The model used for inference or training
- *Services*: The services exposed to other nodes at runtime
- *Partitioners*: How the node decides when to pass intermediary inference data to another node
- *Records*: How the observer compiles and saves benchmarks upon completion
- *Tasks*: Which tasks may be assigned to participating nodes

Each of these subsections provide base modules for test designers to extend, making it straightforward to implement custom behavior that is compatible with the test-bed. User classes can be defined in the base module itself, or as a standalone module within the corresponding subsection; the new classes are integrated into the `experiment_design` package automatically for either method, which means they can be assigned to nodes in an experiment manifest with no further setup required.

*3.2.2 App API Package.* The test-bed itself is driven by an API package written in Python. The user interface is implemented in the main application file (`app.py`), which simply interprets the commands and delegates them to one of three distinct sub-modules in the API:

- *The Setup Module*: Responsible for overseeing device setup and health checks, managing user preferences, and configuration files.
- *The Device Module*: Engages in the discovery and configuration of participating devices, facilitates network communication, and conducts remote device health assessments.

- *The Experiment Module*: Administers the user's experiment specifications, controls experiment run-times, interprets and applies experiment parameters, and saves performance metrics.

Though decoupled from the test-bed's experiment design features, the API is responsible for orchestrating multiple network devices, each of which may use different architectures with different requirements for their runtime environments. At a high level, the primary function of the API is to keep the experiment designer insulated from many of the low-level variables inherent in running a distributed computing experiment so that experiments can be designed and run in a platform-agnostic manner.

The test designer's custom implementations are distributed to each node over SSH within a temporary environment, and an RPC service inheriting those custom implementations is deployed automatically. Meanwhile, the API also deploys an Observer node to the local machine, waits for a "ready" signal from each Participating node, then delegates the user-defined tasks to the appropriate nodes. A "start" signal is sent to the Observer, and the API collects and displays updates from each node in real time. When the experiment concludes, the Observer passes the collected benchmark data to the API, and the data is formatted and saved according to the user's specifications. Finally, each participant node "self-destructs", removing the temporary environment from the host machine.

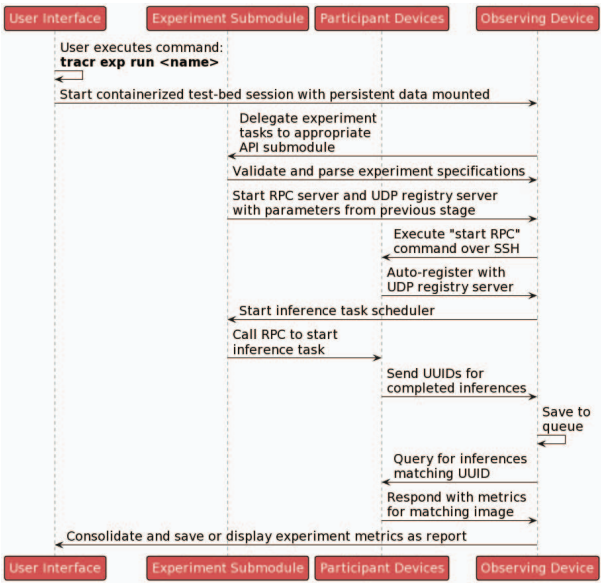


**Figure 2: Sequence Diagram of Experiment Components**

*3.2.3 Test Definitions.* Test cases are defined using a YAML manifest file and code provided by the user to further extend the behavior of the participating nodes that will be involved at experiment run time. Here, we can think of a test case as an experiment with a specific set of parameters applied.

The test-bed repository also contains the *My_Datasets* directory for users to integrate their own datasets that may not be available through the standard avenues. Because these may be used by multiple experiments, they are not stored within the scope of a single experiment.

When a user wishes to run an experiment via the user interface, they will use the `tracr experiment run` command with positional arguments to select which experiment is run, and with what parameters. The job of the manifest file is essentially to set the default experiment parameters at runtime for anything that has not been explicitly overwritten in the `tracr experiment run` command.

## 3.3 Participating Node

*3.3.1 Environment.* The test-bed is deployed by the observer setup procedure within a virtual environment to isolate packages from the existing python interpreter. When the test-bed modules are installed they attempt to gather the latest versions for that platform, but we do not intend to accidentally upgrades external packages on the test device. Inside the environment are three main components; an RPC node for communication, a model wrapper to accept `pytorch` models, and a scheduling service to determine where model splits will occur.

The main script for the environment handles interpretation of the configuration, the running sequence, and links these modules together as required. As a forward pass begins, it loads an image from the specified generator, and creates a unique identifier for the inference it triggers. Once that forward pass is completed, no matter by which participating device, that environment script passes the identifier to the Observer RPC. The Observer RPC then queries each Node for its portion of the inference results, which are removed from the Node to avoid potential memory constraints.

*3.3.2 DNN Wrapper.* The core of each Node is the DNN Wrapper. Our test-bed is capable of testing models in a naive setup, in which we make two assumptions: it can be loaded as a `pytorch` model, and its' layers are only processed once per forward pass. By using this wrapper and a combination of `pytorch` hooks, any `pytorch` model is able to be split apart and benchmarked in the computational offloading test. We take two profiling passes of a wrapped model. The first is performed by `torchinfo`, which we use to acquire information regarding parameters and bytes involved at every depth of the model for use in benchmarking. The second pass is performed by our custom module, which marks layers at a specified depth for splitting, and prepares the logging dictionary. These marked layers are given a pre-hook and post-hook which trigger due to the call implementation in `pytorch`. Both hooks track the progress through the forward pass and record results. This profiling pass also assembles all NN layers for use within the Partition Scheduler (see Eq. 3).

The pre-hook for each layer handles when to inject a provided Tensor input. As pretrained models are not able to be truly split without customization, the layers before the injection point must still be run to reach that point. The pre-hook recognizes this, and provides correctly sized starting data at layer 1. When the injection point is reached, the actual input tensor is swapped into place. Layers that occur before the requested injection point are not tracked in bench-marking. This extends the duration of a test in real-time, but is not included within test results.

**Algorithm 1** Pseudocode for Wrapped Forward Pass

```
initialize sequence values
initialize benchmarking
for all module at depth in model do
    module pre_hook
    module forward pass
    module post_hook
end for
exit handling
finalize benchmarking
```

A post-hook for a layer handles whether and when to exit from the forward pass. Exiting a forward pass early is accomplished by raising and handling an exception at the specified layer. The remaining layers are not required to be run, unlike the extra layers at the entry-point. A pseudo-code example of this structure is provided in Algorithm 1.

Currently bench-marking depth must be correctly specified for accurate results on skip connection models. In ResNet models, we configure the depth of bench-marking to rest at the `BasicBlock` class layer, not its sub-layers. If we went deeper, inference is still performed, but the skip connection is not yet passed to the Cloud Node and it would rely on the dummy data from the inference while it advances to the injection layer.

*3.3.3 Partition Scheduler.* The partition scheduler determines at which point the model will be split during the test. It acquires the NN layers from the DNN Wrapper, and when called will return the layer index it determines a split should occur.

At this time two different partition schedulers are implemented. The first is a simple cyclic partitioner. It starts at the first allowed layer index, repeats that layer index $n$ times, and advances the index one stage until the number of layers defined in the model is reached, repeating infinitely. This is most useful to gather data on each possible split point for a given input across an entire data-set. These results can validate the performance of our next partitioner, a Linear Regression partitioner in the style of Neurosurgeon[7]. The execution time for layer $j$ is approximated using a linear regression model

$$L_j(m_j) = a * m_j + b \tag{1}$$

where $a, b \in R$ and $m_j$ is the number of bytes in a layer $j$.

The linear regression partitioner performs warm-up operations on the Node during startup, and pulls the bench-marking data from these to estimate the execution time using Eq. 1 based on the layer type (e.g., convolution, pooling, etc). To accommodate possible quantization, we compute $m_j$ the number of bytes in a layer $j$ as follows.

$$m_j = \sum_{i=1}^{n} P_i * S_i \tag{2}$$

where $P_i$ = precision of the elements of parameter $i$ in bytes
$S_i$ = number of elements in a parameter $i$
$n$ = number of parameters for the selected layer

The computed $m_j$ are used to build our regression models in Eq. 1. One linear regression model is created for each unique layer type encountered within sequence $k$.

During setup, the Edge Node requests the dictionary of the regression models from the selected Cloud Node to utilizes in execution time estimation. We use $E_j$ or $C_j$ in place of $L_j$ in Eq 3. We use Inequality 3 as the criterion to select the best layer to split. If the inequality is true for a layer $j_c$, starting from layer 1, the index of that layer, $j_c$, is returned as the index a split should occur. If it is false, $j_c$ increments and the inequality is re-evaluated.

$$\frac{m_j}{t_s} + \sum_{j=j_c}^{w} C_j(m_j) < \sum_{j=j_c}^{w} E_j(m_j) \tag{3}$$

where :

$C_j$ = Execution time model of layer j on Cloud node
$E_j$ = Execution time model of layer j on Edge node
$t_s$ = network speed in bytes per second
$w$ = Number of layers in sequence $k$
$j_c$ = candidate layer for split point

Given the assumption that the Cloud device is more capable than the initiating Edge device, if one layer can be completed more rapidly at the Cloud, every following layer will also be completed more quickly. With this, we simplify the criterion to Inequality 4 below.

$$\frac{m_j}{t_s} + C_j(m_j) < E_j(m_j) \tag{4}$$

*3.3.4 Internal RPC Node.* The internal RPC node handles calls from the Observer device for information, as well as from other participating nodes. Servers are instantiated as a singleton object to ensure safety of operation with usage of their contained model. For this baseline configuration, the image generating device registers with the Registry server as an Edge device. Devices that are available to complete an inference register as a Cloud device. Little is different with these except for the internal alias, in anticipation of future work.

RPC methods are written to serialize objects it passes. The underlying package `rpyc` attempts to proxy objects that are passed by reference, a feature that in this case we must avoid. For the bench-marking required we must pass all models and tensors by value to accurately measure timing and avoid overhead. General objects such as the bench-marking dictionaries or small linear models only require standard pickling is performed.

Tensor objects are orders of magnitude larger, and reducing the serialized size is extremely important. When the RPC nodes pass Tensor objects, they are compressed using the `blosc2` python package, which supports serialization of tensors out of the box. As a Cloud RPC device receives a tensor to complete inference on, it begins inference in a thread to avoid blocking the return of the RPC method and disturbing bench-marking on the calling Node.

## 4 AN EXPERIMENT EXAMPLE USING THE TEST-BED

To demonstrate the test-bed's utility, we replicated the results from an experiment described in [7]. In this experiment, inferences are

made using [8] while partitioning at each possible split layer to inspect the differences in overall performance across layers. These results are then compared to the optimal split layer predicted with the Neurosurgeon algorithm. Our experiment was conducted by designing a test case that deploys an edge node to a Raspberry Pi 4B on the local network, while deploying the cloud and observer nodes locally to a consumer GeForce GTX 1050 Ti. The edge node employs the cyclic partitioner to initiate inference tasks at each possible split layer. In this test, the network speed was pinned at 4MB/s to perform at average speeds expected from LTE networks as in Neurosurgeon's results. In our experiment on the test-bed, we observed that the linear regression estimator selected layer 14 consistently in this test setup. To validate this result, the cumulative results from the cyclic partitioning experiment are recorded in Figure 3.

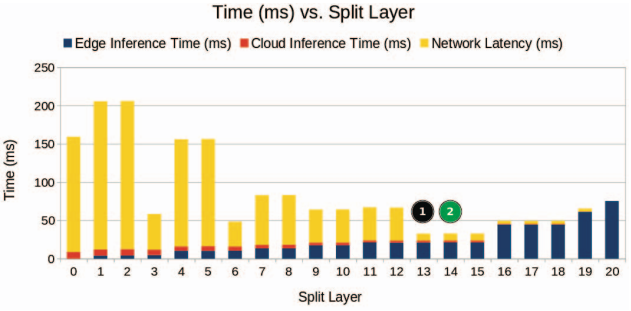


**Figure 3: Total Inference Time by Split Layer**

As shown in Figure 3, we have highlighted with a black "1" the layer selected by the Partition Scheduler described in Section III, and the layer highlighted with the green "2" was reported in [7] using the "Neurosurgeon" algorithm. The current PyTorch implementation of Alexnet slightly differs from the Neurosurgeon implementation. The lowest latency layer, 'pool5' identified there best maps to our layer's 13 & 14, which each perform a portion of the originally implemented pooling. As pooling layers, they are fast to process, and natural bottlenecks due to the reduction in internal size. We believe the selection of either layer is valid for comparison, though a split at 14 would allow cloud processing to begin slightly faster.

## 5 CONCLUSIONS AND FUTURE WORK

Our highest priority for future work is evaluating the possibility of support for skip connections. Skip connections are extremely common, and essential for full depth evaluation of ResNet or Yolo with its feature pyramid. To function in the naive testing scheme the test-bed operates under, users must supply dependency information for the inputs of various layers. This information can be used to ensure the transfer time calculation from all dependant layers at time of completion, rather than just the previous layer. Differently implemented skip connections do not all use consistent storage within the forward pass, so transfer of those outputs is not naively implemented. Scoping of these outputs within the model will also prove difficult to access. One possible solution for testing purposes only is to transfer the starting image in an untimed method, and generating the initial data with that, only injecting the true input to the layer at the specified point.

Second to this is the implementation of layer-wise bottleneck injection. Quantization is well within scope, as it is robustly supported in `pytorch` as is, but on the fly adjustment is our goal to accommodate differing network conditions. Direct manipulation of layers could also allow us to directly link encoders instead of simply adjusting precision.

Lastly, we intend to incorporate the zero-deploy methodology implemented in `rpyc`. Connecting a blank slate device to the testing network and configuring it to act as a participating node with minimal setup will be highly beneficial for larger test networks.

## ACKNOWLEDGMENT

This research project is funded by NSF Award No. 2128341.

## REFERENCES

[1] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. 2018. Mobile cloud computing for computation offloading: Issues and challenges. *Applied computing and informatics* 14, 1 (2018), 1–16.

[2] Juliano S. Assine, J. C. S. Santos Filho, and Eduardo Valle. 2021. Single-Training Collaborative Object Detectors Adaptive to Bandwidth and Computation. arXiv:2105.00591 [cs.CV]

[3] Jiasi Chen and Xukan Ran. 2019. Deep learning with edge computing: A review. *Proc. IEEE* 107, 8 (2019), 1655–1674.

[4] Robert A. Cohen, Hyomin Choi, and Ivan V. Bajic. 2021. Lightweight Compression of Intermediate Neural Network Features for Collaborative Intelligence. *IEEE Open Journal of Circuits and Systems* 2 (2021), 350–362. https://doi.org/10.1109/ojcas.2021.3072884

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[6] Diyi Hu and Bhaskar Krishnamachari. 2020. Fast and Accurate Streaming CNN Inference via Communication Compression on the Edge. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. 157–163. https://doi.org/10.1109/IoTDI49375.2020.00023

[7] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[9] Chunjie Luo, Xiwen He, Jianfeng Zhan, Lei Wang, Wanling Gao, and Jiahui Dai. 2020. Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices. *CoRR* abs/2005.05085 (2020). arXiv:2005.05085 https://arxiv.org/abs/2005.05085

[10] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1396–1401.

[11] Yoshitomo Matsubara and Marco Levorato. 2021. Neural Compression and Filtering for Edge-assisted Real-time Object Detection in Challenged Networks. In *2020 25th International Conference on Pattern Recognition (ICPR)*. 2272–2279. https://doi.org/10.1109/ICPR48806.2021.9412388

[12] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. 2022. Split computing and early exiting for deep learning applications: Survey and research challenges. *Comput. Surveys* 55, 5 (2022), 1–30.

[13] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.