A Dynamic Distributed Scheduler for DNN Inference on the Edge

Fei Hu *
Department of Computer Science
University of Colorado
Boulder, CO, USA
fei.hu@colorado.edu

Shivakant Mishra
Department of Computer Science
university of Colorado
Boulder, CO, USA
mishras@colorado.edu

Abstract—**Edge computing plays a pivotal role in IoT applications that require rapid and secure data processing. However, these applications are typically resource-demanding, and the resources available at the edge are often significantly less than those available in cloud environments. Thus, optimally utilizing these constrained resources is key to fulfilling various application requirements such as low latency, privacy, and cost-effectiveness. Given the dynamic and hybrid characteristics of IoT environments, static scheduling approaches frequently fail to adequately address these complex and varying demands. This paper presents the design, implementation, and evaluation of a dynamic distributed scheduler tailored for DNN inferences on edge computing platforms, based on sophisticated sub-model profiling techniques. This scheduler's core feature is its ability to continuously monitor the state of the IoT infrastructure, dynamically adapting the distribution of computing tasks according to real-time environmental and network conditions. Diverging from previous research that typically segments DNN models into two parts, this study proposes a more granular approach that splits models into multiple segments, thereby maximizing the utilization of diverse edge computing resources. The implementation of this scheduler, including integration with a GPU, demonstrates its effectiveness across various DNN models, highlighting its practical utility in leveraging edge computing capabilities.**

Index Terms—**Dynamic Distributed Scheduler, Parallel Computing, Edge Computing, DNN Models, Device Profile**

I. Introduction

Edge AI systems are increasingly recognized as a vital strategy for managing the rising demands for AI workloads by decentralizing processing closer to data sources, thereby minimizing latency [1]. A principal obstacle in deploying AI applications at the edge is their substantial computational demands relative to the more limited computing resources typically available on edge servers compared to cloud environments. Traditionally, developers have tackled this by dispersing the AI application's computational tasks among various nodes within the edge network, which might include both rudimentary sensing devices and more sophisticated processors at the edge server. Nonetheless, this approach introduces the trade-off of increased data transfer times between nodes against the benefits of parallel processing. Achieving optimal performance thus requires a nuanced balance between enhancing processing speed through task distribution and mitigating the latency introduced by data transfers. Three predominant challenges in IoT settings further complicate this balancing act: (1) Network conditions are not static but vary dynamically, affecting the delays in data transfers; (2) The computational load on nodes can change based on dynamic environmental factors, affecting task processing times; and (3) Edge computing nodes exhibit significant diversity in their computational, storage, and networking capabilities [2]–[4]

This research is supported by grants from NSF.

Building on the work presented in [3], which addresses the scheduling challenges for container-based AI applications, this paper extends the investigation to scheduling issues specifically for DNN models.

For heavyweight DNN applications, edge devices—limited in computing resources— sometimes cannot process the entire application independently. Current research addresses this challenge by partitioning the DNN model into sub-models, enabling edge devices to collaborate on a single inference task through parallel computing. This paper focuses on the challenge of optimally partitioning DNN models and proposes a series of solutions to tackle this issue, as detailed in the subsequent points.

1) Characterizing Layers in DNN Models: Delve into the specifics of different layers within DNN models, aiming to understand their roles, functionalities, and impacts on the network's performance and computational demands.
2) Profiling of DNN sub-models: To address heterogeneity among different computing devices, each device in the IoT environment is profiled for different DNN models under different computing loads to enable the scheduler to make optimal scheduling. Compared to container-based applications, DNN models can be divided into several sub-models. This profiling targets those potential sub-models for efficient segmentation.
3) Optimal Scheduling: In the IoT system, all device profiling information is shared with and updated on the edge server, where the master scheduler makes global optimal scheduling decisions. The guiding principle is to process tasks locally if the end devices have the capacity. If not, all computing nodes are leveraged cooperatively to handle a single DNN inference.

We have developed and extensively tested a prototype of a dynamic, distributed scheduler tailored for DNN models across various computing scenarios. The evaluation results show that this scheduler significantly surpasses existing edge scheduling solutions in meeting diverse application demands. Specifically, this paper offers the following key contributions:

- The proposed dynamic distributed scheduler effectively tackles the challenges posed by device heterogeneity and the dynamically changing compute and network conditions typical in edge IoT environments. It achieves this by dynamically segmenting DNN models and adapting task distribution in response to prevailing operational conditions.
- To the best of our knowledge, the proposed sub-model profiling represents the first initiative to apply an optimal DNN model layer partitioning method. This approach not only reduces the workload associated with pre-evaluation but is also practical for application across a wide range of DNN models.
- The paper presents a prototype implementation of the proposed scheduler and conducts a thorough performance evaluation across various operating conditions. The experimental results demonstrate that this distributed architecture effectively enhances edge computing efficiency through parallel processing. Building on the research presented in paper [3], we have developed a dynamic distributed scheduler that supports a wide range of AI applications for edge computing.

II. Related Work

A fractional offloading strategy leveraging the distinctive layered structure of Deep Neural Networks (DNNs) can be utilized. This approach, called DNN model partitioning, divides the model so that some layers are processed on the device and others on the edge server or cloud, potentially reducing latency by harnessing the computational power of additional edge devices [5].

Identifying the optimal partition point in a DNN involves three key steps: first, assessing and modeling resource consumption and data exchange between layers; second, estimating total operational costs based on layer configurations and network bandwidth; and third, selecting the best partition point, considering factors like delay and energy requirements. [6], [7].

Horizontal partitioning of the Deep Learning (DL) model, distributing it across the end, edge, and cloud layers, is the most prevalent method of segmentation. The difficulty rests in smartly determining where to partition the DNN model. COMET offloads a thread solely based on whether its execution time surpasses a predefined threshold, disregarding other factors like data transfer volume or wireless network availability [9]. Odessa, on the other hand, bases its computation partitioning decisions on the execution time and data requirements of only a portion of the function, neglecting the broader context of the entire application [10]. CloneCloud applies the same offloading criteria across all instances of a given function [11]. In contrast, MAUI represents an improvement in offloading decision-making by evaluating each function invocation individually and taking the entire application into account when determining which functions to offload [8]. Paper [12] introduces an on-demand, low-latency inference framework that optimizes both the model partition strategy, tailored to the disparate computational capacities of mobile devices and edge servers, and the early exit strategy, adapted to complex network conditions. Various strategies have been devised to distribute a pre-trained DNN across multiple mobile devices, aiming to speed up DNN inference on these devices [13]–[16]. Bhardwaj et al. expanded on this by incorporating memory and communication costs into the distributed inference framework. To tackle these challenges, they suggested model compression techniques and a network science-based algorithm for partitioning knowledge [17], [18]. An alternative approach to model segmentation involves vertical partitioning [19].

However, current research reveals several shortcomings: 1) Lack of consideration for crucial constraints such as latency, energy, and privacy [9], [10]; 2) Insufficient flexibility for dynamic edge environments [11]; 3) Inadequacy in distributing DNN applications across end devices, edge devices, and the cloud [13], [14], [17]; 4) Ineffectiveness in determining partition points in architectures with multiple end devices [6].

III. Scheduler Design

Deep Neural Networks (DNNs) are structured as directed graphs, where each node functions as a neuron processing incoming data to produce outputs. An example depicted in 1 shows a 5-layer DNN used for image classification, with computational flow directed from left to right. The connections between neurons map the data flow, forming layers of neurons that perform identical functions on different input segments. During a DNN's forward pass, the output from one layer serves as the input for the next, with the network's depth determined by its total number of layers.

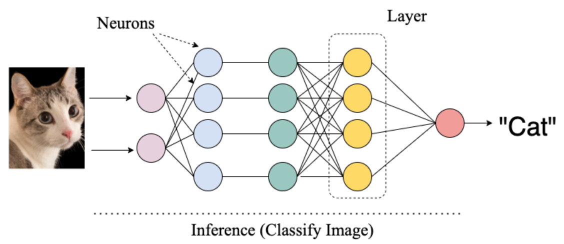

Fig. 1: A 5-layer DNN Classifies Inference

For applications like face detection, which are relatively lightweight, the task can be processed entirely locally or offloaded to an edge server. However, for DNN models, a fractional offloading approach is more feasible in edge environments with limited computing resources. In this partitioning method, some layers are processed on end devices while others are handled by the edge server, as demonstrated in Fig. 2. This example of horizontal partitioning shows a 5-layer DNN model where node 1 processes the first two layers locally before sending the results to the edge server, which may further distribute tasks to other nodes like node 2.

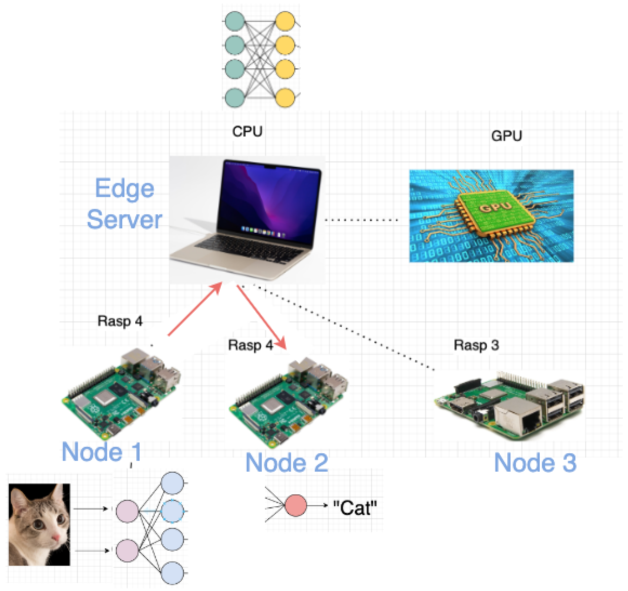

Fig. 2: A DNN Inference Distribution Example

Compared to full application offloading, the main challenge in partial offloading is identifying the optimal points for partitioning the layers. This requires a careful balance between minimizing latency and maximizing computational efficiency across devices. This paper proposes a design of a dynamic distributed scheduler for performance improvement.

A. Scheduler Design Principles

Edge IoT environments are characterized by their dynamic and hybrid nature, encompassing a diverse spectrum of devices from resource-constrained units to robust edge servers. The variability in communication and computational demands, driven by real-time events, necessitates that schedulers dynamically adjust to these changes during runtime. However, this requirement for dynamic adaptation mandates continuous monitoring of current computing and networking conditions, which introduces additional performance overhead. Thus, achieving a balance between this overhead and the performance improvements from near-optimal scheduling—enabled by accurate, real-time updates on compute and network states—is crucial for effective task scheduling and enhanced overall application performance.

In the context of DNN models, with global information, the distributed scheduling coordinator makes near-optimal decisions based on the local compute principle at the end devices. This approach prioritizes executing tasks on the end device possessing the necessary data, provided it can satisfy all application constraints such as real-time latency requirements. This principle ensures tasks are processed on the device managing the IoT sensor that requires the data. Conversely, if an end device cannot meet all application constraints through local execution, the task may be entirely or partially offloaded to other end devices or the edge server, ensuring efficient and timely task execution.

B. Device Profiling

Central to our scheduling methodology is the role of edge server scheduling, which is crucial for optimizing overall system performance. The edge server not only determines which device will execute a task but also assesses the computational capacity of the end device, specifically how many layers it can effectively process.

To address these challenges, we propose a proactive solution based on the pre-evaluation of sub-models. This approach involves estimating the processing time between successive layers on each device. For instance, we might calculate the time required for a sub-model to execute from Pooling 1 to Pooling 2 in VGG models. This information enables the edge server to make informed scheduling decisions that comply with predefined time constraints.

However, a significant challenge arises from the impracticality of obtaining processing times for all sub-model configurations. We will explore this issue further and discuss our strategies for overcoming it in Section IV.

C. Architecture Workflow

Algorithm 1 The Dynamic Distributed Scheduler Algorithm

```
Input: profile tables of all computing nodes, time constraint $T$
Output: number of frames meeting time constraint, video with augmented reality effect
1: $N$                          ▷ total layer number of the DNN model
2: $R_{ab}$                     ▷ time of sub-model from layer a to b on Rasp
3: $E_{ij}$                     ▷ running of layer i to j on the edge server
4: $G_{mnL}$                    ▷ time of layer m to n on GPU with workload L
5: $T_{RG}$                     ▷ transmission from Rasp to GPU
6: $T_{GE}$                     ▷ transmission from GPU to edge server
7: if $R_{0N} < T$ then
8:     processing locally on Raspberry Pi
9: else
10:    for do each $b$ in $N...1$
11:        if $R_{ab} + G_{bNL} + T_{RG} < T$ then
12:            layer $a$ $b$ on Rasp, $b+1$ $N$ on GPU
13:        else
14:            if $R_{ab} + G_{bjL} + E_{jN} + T_{RG} + T_G < T$ then
15:                layer $a$ $b$ on Rasp, $b+1$ $j$ on GPU, $j+1$ $N$ on edge server
16:            end if
17:        end if
18:    end for
19: end if
```

When an end device such as a Raspberry Pi (R1) receives a classification request, it forwards this request to the edge server. As all end devices regularly update their profiles with detailed processing and transmission times, the edge server leverages this comprehensive global information to make optimal scheduling decisions and provide timely responses.

The system configuration, comprising a Raspberry Pi (R1), a GPU, and an edge server, is detailed in Table II, and the corresponding scheduling algorithm is outlined in Algorithm 1. If R1 is capable of processing the entire inference task locally, it proceeds without further coordination. If, however, R1 cannot meet the latency requirements independently, the scheduler determines an optimal cutting point to maximize the layers processed on R1 while adhering to time constraints. The GPU, generally more powerful than R1, is tasked with processing the subsequent layers. In this setup, the edge server functions primarily as a coordinator for scheduling rather than as a computational resource for DNN tasks. Nonetheless, in scenarios where the GPU reaches capacity, it may partition the model further, delegating the processing of additional layers to the edge server to maintain efficiency.

In light of the algorithm, the scheduling overhead can become significant for DNN models composed of numerous layers. Pooling layers, however, represent an advantageous cutting point, as they significantly reduce both processing time and output data size, thereby minimizing data volume and enhancing processing speed. Consequently, we advocate for the use of pooling layers as strategic cutting points. Detailed discussions and justifications for this approach will be elaborated in Section IV.

IV. DNN Model Evaluation

we propose to solve the DNN partition problem in the following steps 1) Characterizing Layers in DNN Models and 2)Pre-evaluation of DNN Model.

A. *Characterizing Layers in DNN Models*

The various types of layers in today's DNN models include:

1) Fully-connected Layer (fc)- Every neuron in a fully connected layer is linked to all neurons in its previous layer. This layer calculates the weighted sum of the inputs using predetermined weights
2) Convolution & Local Layer (conv, local) - In convolution and local layers, an image is processed through convolution with learned filters to generate feature maps. The variation among these layers comes from the size of their input feature maps, the dimensions and quantity of the filters, and the stride at which these filters are applied.
3) Pooling Layer (pool) - Pooling layers group features together by applying a predefined function, such as max or average, to regions of input feature maps. The key differences in these layers lie in the size of their input, the dimensions of the pooling region, and the stride at which pooling is executed.
4) Activation Layer (act) - Activation layers introduce non-linearity to neural networks by applying specific functions to each input individually, outputting data in equal volume.

Other layer types include normalization layer(norm), softmax layer (softmax), argmax layer (argmax),dropout layer (dropout), etc.

Fully-connected layers introduce high latency, while convolution and pooling layers at the network's front end show shorter latency. As convolution layers increase and pooling layers decrease data size, this reduction progresses toward the back end where fully-connected layers are situated. This dynamic presents a unique opportunity for computational partitioning in the middle of the DNN, optimizing between the mobile and cloud environments.

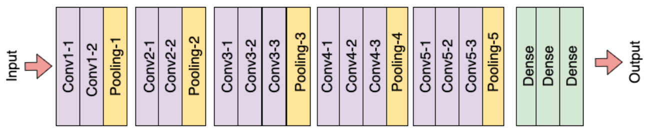

Fig. 3: VGG16 Architecture

Take VGG16 as an example [20]. VGG16 is a convolutional neural network designed for object detection and classification, achieving an accuracy of 92.7%. It consists of thirteen convolutional layers, five Max Pooling layers, and three Dense layers, totaling 21 layers, as Figure 3 shows. However, only sixteen of these layers contain learnable parameters. VGG16 accepts input tensors of size 224 by 224, each with 3 RGB channels.

TABLE I: Output Data Size at Each Layer

Layer	input1	conv1-1	conv1-2	pool1	conv2-1	conv2-2	pool2	...
Data Size (MB)	0.57	12.25	12.25	3.06	6.13	6.13	1.53	...

B. *Pre-evaluation of DNN Model*

We propose to evaluate sub-model-based profiling at each pooling layer, which includes

1) Data Size Variations - Evaluate the output data size for each pooling layer on a specific device. In neural networks, convolution layers initially expand data which is subsequently reduced by pooling layers, progressively diminishing the data size through the network. As a result, the data size in the final layers is smaller than the original input.
2) sub-model Latency - Evaluate the sub-model running time at each pooling layer, as observations indicate that incorporating a pooling layer typically results in reduced processing times for sub-models.

A shorter processing time and reduced output data size make a pooling layer an ideal cutting point, as it both minimizes data volume and accelerates processing. Additional evidence will be provided to support this assertion.

We designated a 1.9MB image as the input and observed the data size changes across each layer, with a subset of the results presented in Table I. The initial layers involve a pattern where convolutional layers increase data volume, followed by pooling layers that decrease it, effectively reducing the overall data size progressively. For instance, the data size at the input layer stands at 0.57MB, which escalates to 12.25MB post-Conv1. This increase is attributed to the Conv-1 layer's utilization of 64 filters, while subsequent layers—Conv-2 with 128 filters, Conv-3 with 256 filters, and Conv-4 and Conv-5 each deploying 512 filters—substantially enhance the depth relative to the original input. Each pooling operation, employing a 2×2 kernel with a stride of 2, diminishes the height and width of the feature maps, exemplified by Pooling1 which reduces the data size from 12.25MB to 3.06MB.

To fully leverage the computing resources of edge devices, it is crucial to balance data transfer times with computation latency. Therefore, we evaluated the sub-model latency on an edge server, with system specifications detailed in Table II, and part of results are shown in Table III. Sub-model latency measures the processing time from the input layer to a given layer, focusing particularly on the last convolution layer of each block and the subsequent pooling layer. Our findings indicate that deeper layers generally exhibit increased processing times. Interestingly, we observed that the inclusion of a pooling layer typically reduces running time. For instance, the running time after the first convolution layer (Conv1-2) is 97.33 ms, while the subsequent pooling layer (Pooling1) decreases to 96.73 ms, with later blocks following this trend.

TABLE II: System Configuration

Component Name	**Specifications**
Edge Server	2.3GHz Dual-Core Intel Core i5, 8 GB RAM, 256GB Disk
Raspberry Pi	Quad core Cortex-A72 (ARM v8) 64-bit 8GB RAM, 1.8GHz Clock Speed
GPU	128-Core Maxwell GPU, Quad-Core ARM @ 1.43GHz CPU, 4GB RAM

TABLE III: Sub-Model Latency

Layer	Conv1-2	pool1	Conv2-2	pool2	Conv3-3	pool3	Conv4-3	pool4	Conv5-3	pool5
Time (ms)	97.33	96.73	145.27	143.10	237.71	224.46	294.06	291.06	319.16	318.02

Adding a pooling layer typically reduces sub-model processing time due to two factors: 1) Data Movement and Copying: Splitting a model introduces overhead from data transfer between hardware components or within the GPU, which can negate computational savings from fewer operations due to pooling. 2) Model Caching and Optimization: Deep learning frameworks optimize computational graphs more effectively when models are run as a whole rather than in parts. These optimizations include layer operations and memory usage across the entire model, and splitting the model can disrupt this. Thus, a pooling layer serves as an effective cutting point because it decreases data size and enhances processing efficiency.

V. Evaluation

We established various scenarios to assess the effectiveness of the DNN layer partitioning and dynamic scheduling algorithm. We began with Case 1, using a Raspberry Pi as the edge device and an edge server, and progressed to Case 2, which incorporates GPU utilization.

A. *Case1: Raspberry Pi and Edge Server*

In this scenario, the Raspberry Pi receives requests for 50 VGG16 inferences and forwards them to the edge server. The edge server, serving as the coordinator with comprehensive global information, makes scheduling decisions for each inference. It determines whether to split the model and at which point to execute the split, then communicates these decisions back to the Raspberry Pi. If the model is split, the latter half will be offloaded to the edge server for further processing.

We first determined the success rate, defined as the proportion of inferences completed within the time constraints. The results, presented in Table IV, show that our dynamic distribution scheduler achieved a success rate of 98%. This significantly outperforms the Neurosurgeon algorithm, which achieved a success rate of 82%, underscoring the superior efficiency of our scheduler.

TABLE IV: Success Rate

Algorithm	Dynamic Scheduler	Neurosurgeon
Success Rate	98%	82%

We also assessed the edge device utilization rate, which reflects the proportion of layers processed on the endpoint device. The findings, detailed in Table V, reveal that the utilization rate for our dynamic distributed scheduler stands at 33%, which is 10 times higher than that achieved by the Neurosurgeon algorithm.

TABLE V: Edge Device Utilization Rate

Algorithm	Dynamic Scheduler	Neurosurgeon
Utilization Rate	33%	4%

B. *Case2.1: Rasp and GPU*

GPUs excel in handling computer graphics and image processing due to their highly parallel architecture, making them superior to general-purpose CPUs for algorithms that process large data blocks simultaneously. Recently, GPUs have become increasingly popular for demanding computational tasks in areas like deep learning and data mining.

We incorporated a GPU to highlight the effectiveness of our dynamic distributed scheduler in this scenario. The Raspberry Pi receives requests for 50 DNN inferences and forwards them to the edge server. Acting as the coordinator with comprehensive global information, the edge server makes scheduling decisions for each inference, including whether to split the model and at which layer to execute the split, and then relays these decisions back to the Raspberry Pi. If the model is split, the GPU instead of the edge server functions as an additional processing node.

C. *Case2.2: Rasp, GPU, and Edge Server*

This scenario mirrors case 2.1, with the distinction that the edge server also functions as a computing node. If the GPU is

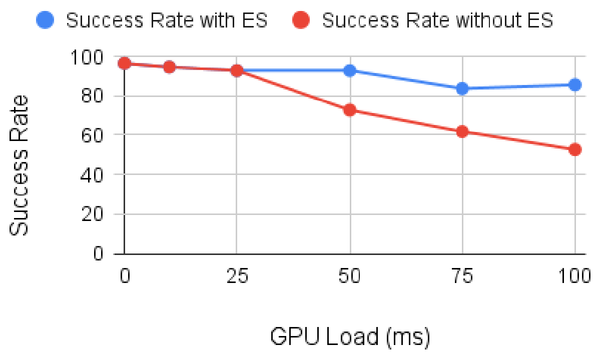

Fig. 4: Success Rate with Variable GPU Load

overloaded and unable to handle the current task, the model may either be offloaded from the GPU to the edge server or the GPU may split the model at a specific layer, offloading the subsequent layers to the edge server for processing. All decisions are made by the edge server when the Raspberry Pi sends a request to it.

Figure 4 presents the experimental results for scenarios 2.1 and 2.2, under varying GPU loads ranging from 0ms to 100ms. As the GPU load increases, the DNN processing capacity diminishes. The blue line represents the system's success rate when the edge server functions as a computing node, while the red line depicts the success rate when the edge server solely acts as a scheduling coordinator. As illustrated by the blue line, an increase in GPU load correlates with a decrease in success rate. This decline indicates that when the GPU is burdened with additional tasks, its ability to handle parallel computations is compromised, leading to longer processing times and lower success rates. The red line exhibits a similar pattern. A comparison between the two lines reveals that as the GPU load escalates, the edge server assumes some tasks that the overloaded GPU cannot process promptly, thereby fulfilling the time constraints.

VI. Conclusion

this paper introduces a dynamic distributed scheduler optimized for DNN inferences in edge computing environments. Our approach diverges from traditional static scheduling by adapting in real-time to changes in computing and network states, thus enhancing resource utilization. Key contributions of this work include 1) A novel model partitioning strategy that segments DNN models into multiple sub-models, allowing for more flexible and efficient use of edge computing resources. 2) The introduction of sub-model profiling, a pioneering method that optimizes DNN layer partitioning to reduce pre-evaluation work and adapt to various architectures. 3) A prototype implementation of the scheduler integrated with a GPU, extensively tested to demonstrate superior performance over existing edge scheduling solutions. This research enhances the capabilities of edge computing frameworks to meet the stringent requirements of modern IoT applications, establishing a foundation for future advancements in distributed computing.

References

[1] W. Li, and M. Liewig, "A survey of AI accelerators for edge environment," In Trends and Innovations in Information Systems and Technologies: Volume 2 8 (pp. 35-44). Springer International Publishing, 2020.

[2] T. Nepusz, and T. Vicsek, "Controlling edge dynamics in complex networks," Nature Physics, 8(7), pp.568-573, 2012.

[3] F. Hu, K. Mehta, S. Mishra, and M. AlMutawa, "A Dynamic Distributed Scheduler for Computing on the Edge," arXiv preprint arXiv:2308.06806 (2023).

[4] F. Hu, K. Mehta, S. Mishra, and M. AlMutawa. "Distributed Edge AI Systems." In Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing, pp. 1-6. 2023.

[5] J. Chen, and X. Ran, "Deep learning with edge computing: A review," Proceedings of the IEEE 107, no. 8 (2019): 1655-1674.

[6] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," ACM SIGARCH Computer Architecture News 45, no. 1 (2017): 615-629.

[7] Z. Zhao, Z. Jiang, N. Ling, X. Shuai, and G. Xing, "ECRT: An edge computing system for real-time image-based object tracking," In Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems, pp. 394-395. 2018.

[8] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, et al, "Maui: making smartphones last longer with code offload," In Proceedings of the 8th international conference on Mobile systems, applications, and services, pp. 49-62. 2010.

[9] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X Chen, "COMET: Code offload by migrating execution transparently," In 10th USENIX symposium on operating systems design and implementation (OSDI 12), pp. 93-106. 2012.

[10] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," In Proceedings of the 9th international conference on Mobile systems, applications, and services, pp. 43-56. 2011.

[11] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," In Proceedings of the sixth conference on Computer systems, pp. 301-314. 2011.

[12] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," IEEE Transactions on Wireless Communications 19, no. 1 (2019): 447-457.

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

[14] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37, no. 11 (2018): 2348-2359.

[15] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "SPINN: synergistic progressive inference of neural networks over device and cloud," In Proceedings of the 26th annual international conference on mobile computing and networking, pp. 1-15. 2020.

[16] X. Zhang, Mo. Mounesan, and S. Debroy, "Effect-dnn: Energy-efficient edge framework for real-time dnn inference," In 2023 IEEE 24th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), pp. 10-20. IEEE, 2023.

[17] K. Bhardwaj, C. Lin, A.Sartor, and R. Marculescu, "Memory-and communication-aware model compression for distributed deep learning inference on IoT," ACM Transactions on Embedded Computing Systems (TECS) 18, no. 5s (2019): 1-22.

[18] Y. Shi, K. Yang, T. Jiang, J. Zhang, and K. B. Letaief, "Communication-efficient edge AI: Algorithms and systems," IEEE Communications Surveys and Tutorials 22, no. 4 (2020): 2167-2191.

[19] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37, no. 11 (2018): 2348-2359.

[20] S. Mascarenhas, and M. Agarwal, "A comparison between VGG16, VGG19 and ResNet50 architecture frameworks for Image Classification," In 2021 IEEE International conference on disruptive technologies for multi-disciplinary research and applications (CENTCON), vol. 1, pp. 96-99.