AoDNN: An Auto-Offloading Approach to Optimize Deep Inference for Fostering Mobile Web

Yakun Huang, Xiuquan Qiao, Schahram Dustdar, Yan Li

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796763

***Abstract*— Employing today's deep neural network (DNN) into the cross-platform web with an offloading way has been a promising means to alleviate the tension between intensive inference and limited computing resources. However, it is still challenging to directly leverage the distributed DNN execution into web apps with the following limitations, including (1) how special computing tasks such as DNN inference can provide fine-grained and efficient offloading in the inefficient JavaScript-based environment? (2) lacking the ability to balance the latency and mobile energy to partition the inference facing various web applications' requirements. (3) and ignoring that DNN inference is vulnerable to the operating environment and mobile devices' computing capability, especially dedicated web apps. This paper designs AoDNN, an automatic offloading framework to orchestrate the DNN inference across the mobile web and the edge server, with three main contributions. First, we design the DNN offloading based on providing a snapshot mechanism and use multi-threads to monitor dynamic contexts, partition decision, trigger offloading, etc. Second, we provide a learning-based latency and mobile energy prediction framework for supporting various web browsers and platforms. Third, we establish a multi-objective optimization to solve the optimal partition by balancing the latency and mobile energy.**

I. Introduction

The web is becoming the most viable cross-platform technology and ubiquitous platform, reducing the costs of deploying services across a wide range of devices and environments [1], [2]. More than 80% of smartphone apps adopt a hybrid development and support web-based apps using the system's embedded WebView [3]. However, enabling computation-intensive artificial intelligence (AI) services such as deep learning on the cross-platform web is still in its fancy and has been numerous complaints, including: (i) unbearable model loading latency [4], slow response [5], and substantial energy consumption; (ii) the devices have various inference performance of the same service due to the enormous diversity of hardware and software specifications; (iii) the lack of optimized low-level APIs and instant loading mechanism [1], [6], [7]. In this work, we focus on optimizing deep learning inference for fostering mobile web with an auto-offloading mechanism and enhancing WebAI capabilities for web apps.

Yakun Huang and Xiuquan Qiao are with State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, 100876, China. Email:{ykhuang, qiaoxq}@bupt.edu.cn. Xiuquan Qiao is the corresponding author. Schahram Dustdar is with the Distributed Systems Group, Technische Universität Wien, Vienna, 1040, Austria. E-mail:dustdar@dsg.tuwien.ac.at. Yan Li is with the Shanxi Transportation Planning Survey and Design Institute Co., LTD., Taiyuan, 030012, China. E-mail:58093797@qq.com. This research was supported in part by the National Key R&D Program of China under Grant 2018YFE0205503, in part by the Funds for International Cooperation and Exchange of NSFC under Grant 61720106007, in part by the 111 Project under Grant B18008.

There are many kinds of research concentrated on mobile offloading, including (i) conventional solutions that offload JavaScript codes, which require developers to analyze, annotate and identify offloading codes and tasks manually [8]–[11]. This approach mainly includes the Code analyzer and the Migrator. Code analyzer provides the profiler and static analysis to orchestrate the intensive functions and codes between the web and the server. The Migrator provides code migration and JavaScript runtime context synchronization. However, this approach is coarse-grained and inefficient for extending to deep inference, which does not involve analysis and offloading strategies for DNN characteristics. Also, it lacks the use of more efficient multi-threaded web workers to enable energy-efficient DNN offloading; (ii) the second approach delves into the dynamic partitioning and offloading strategies for implementing DNN inference across resource-constrained mobile devices, the edge, and the cloud. This promising approach can be classified as partition-offloading execution [12]–[14] and collaborative execution, which exploring optimal partition of DNN layers and distributing them to various computing resources, or training multi-branches to provide an early exit to reduce needless inference [15]–[20]. However, these efforts focus more on optimal DNN partitioning while lacking exploring the differences and features implementing DNN offloading on the mobile web and native device. Typically, [18], [19], [21] provide simple validation of their collaborative inference by packing API interfaces for the web services.

To enable DNN inference offloading for the mobile web and relieve the burdens of web developers, we propose AoDNN, an auto-offloading framework to distribute the DNN inference across the edge and ubiquitous web-supported devices. AoDNN quickly provides inference migration and recovery by employing a snapshot-based mechanism to answer three critical questions, including "what-to-offload", "when-to-offload", and "how-to-offload" for the mobile web. Nevertheless, there are three critical challenges to implementing the AoDNN.

- **Inefficient JavaScript environment and DNN characteristics lead to web developers having to spend lots of interventions and efforts to enable offloading.** Existing offloading frameworks run custom function code offloading written in Node.js (JavaScript for web apps), Python, or C++ (towards native apps) based on a Function-as-a-Service (FaaS) [22]. They require the professional analysis of web developers to obtain an optimal offloading strategy and cannot be applied directly to offloading DNN inference on mobile web apps. Besides, web apps usually require a

third-party DNN inference library with high encapsulation APIs to execute DNNs, which is hard to distribute due to complex DNN layers, intermediate outputs, and current executing status.

- **The huge variability of hardware and software makes the linear regression for predicting inference latency and mobile energy unavailable.** The same DNN has different layer inference performance between the web apps and the native apps with the same device. Also, we observe that it has weak adaptability on a wide type of DNNs. These altogether indicate that executing prediction models with a better fitting ability and better performance is fundamental to realize optimal DNN inference offloading for web apps.
- **Resource-hungry mobile web apps have stricter requirements to balance the latency and mobile energy than native apps.** Existing methods using single-objective optimization, either latency or mobile energy, cannot meet the demand of achieving a balance among multiple indicators such as latency, mobile energy, accuracy, etc. More importantly, it is necessary to design adaptive DNN partitioning that considers the dynamic contexts, low complexity, and high efficiency.

To address the first challenge, we explore the nature of JavaScript computation offloading on the mobile web platform and propose a snapshot-based offloading framework for fast inference packing, forwarding, and resuming. Specifically, AoDNN uses a multi-threaded web worker technology to control and schedule the DNN inference offloading. We also detail three essential aspects of web browsers in offloading DNN inference, i.e., "what-to-offload", "when-to-offload", and "how-to-offload". Meanwhile, we leverage Node.js as the operating environment for the edge to seamless recovery task snapshots, smoothly execute the DNN inference, and timely respond to web apps. *To address the second challenge* and answer "what-to-offload", we propose the learning-based latency and mobile energy prediction framework. It considers the DNN characteristics and combines the unknown computing capability of mobile devices, running status, and software. Primarily, we first learn polynomial prediction models aiming at the native platform. Then, the multi-type prediction models are further provided for more accurate prediction models, reducing the fitting error caused by DNN running in different stages of web apps. It also reduces the difficulty of fitting a learning prediction model. *To address the third challenge*, we establish a lightweight multi-objective optimization for efficiently making partitioning decisions during the online offloading. It has the optimization objective of both latency and mobile energy, which also can be expanded to other metrics for web apps with extremely limited resources and operating environments. Our adaptive partitioning and offloading mechanism can change the optimal offloading strategy according to the dynamic contexts by real-time monitoring and complete the elastic DNN inference.

We implement AoDNN and generate standard JavaScript manifest files for online offloading. Extensive experiments on typical datasets and DNNs illustrate that proposed learning-based prediction framework performs better than baselines, especially those that ignore the computing capability, running status, and characteristics of web apps. Also, AoDNN on MobileNet improves the latency by 1.8x on average and up to 20.4x, reduces mobile energy by 22.5% on average and up to 32.9%, and improves the throughput by 1.2x on average and up to 2.4x. The key contributions are as follows:

- AoDNN explores automatic offloading approach towards web apps that can seamlessly offload intensive DNN inference between the mobile web and the edge, fully considering DNN characteristics and dynamic web environments.
- We propose exclusive latency and mobile energy prediction models, considering the DNN characteristics, used device, running status, and then further provide more accurate predictions for various web browsers and platforms.
- We establish a multi-objective optimization rather than a single objective to provide low latency and low mobile energy offloading scheme for leveraging distributed DNN, which can quickly expand more metrics for web apps.

II. Design of Proposed Framework

In this section, we introduce the design of the AoDNN approach in Fig. 1, detailing how to provide automatic and seamless offloading of DNN inference, including "*what-to-offload*", "*when-to-offload*", and "*how-to-offload*".

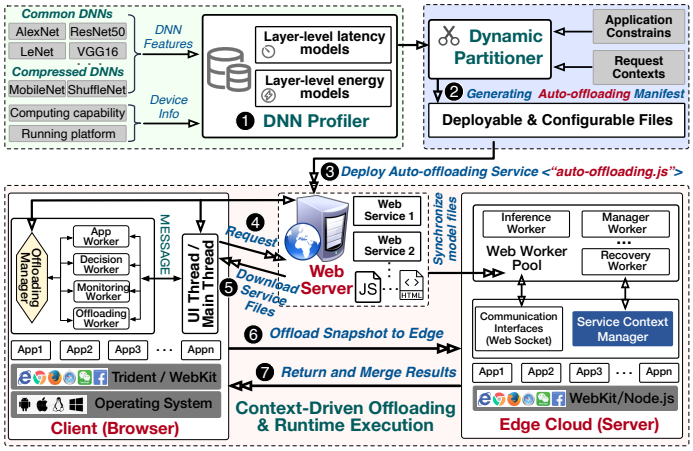


Fig. 1. System overview of DNN auto-offloading for the mobile web.

A. Overview of AoDNN

AoDNN mainly includes DNN Profiler (*what-to-offload*) in the offline phase, DNN Partitioner (*when-to-offload*), and Context-Driven Offloading & Runtime Execution (*how-to-offload*) in the online phase.

DNN Profiler (in-depth analysis of inference for "*what-to-offload*"). This module makes in-depth inference profiling of different common DNNs from the fine-grained layer inference. It provides inference prediction of DNN layers when executed in different browsers and devices. Accurate DNN layer inference analysis provides an essential foundation for DNN partitioning and automatic offloading decisions in the online

phase. In contrast to the traditional approach of using linear regression, which only considers the layer features, DNN Profiler provides a platform-aware prediction framework to predict the layer inference latency and mobile energy. It combines the layer structure, executed platform, and operating environment to propose different polynomial prediction models. Thus, it can fit multiple types of DNN layers and further enhance the effectiveness of the partitioning.

DNN Partitioner. This component provides how to generate manifest files deployed and configured for the web service to offload DNN inference automatically. Besides, dynamic partitioning and offloading execution based on the real-time contexts and environment is also the pivotal content. Typically, the generated manifest files are deployed to application servers that provide standard web services, providing DNN inference offload services without developer intervention. The automatic offloading mechanism of the DNN Partitioner provides optimal partitioning and execution by using DNN Profiler to estimate inference latency and mobile energy without pre-executing the deployed DNNs.

Context-Driven Offloading & Runtime Execution. It tells how AoDNN implements adaptive offloading and distributed inference between the mobile web and the edge (explaining "*how-to-offload*"). The key process includes the browser initiating a DNN computation request to the web server and downloading service files such as model, computation logic diagram, and "auto-offloading.js" required. Then, the browser decides whether to trigger DNN offloading by executing the auto-offloading manifest files. Once the offloading is triggered, the packaged offloading snapshot is synchronized to the server through the WebSocket communication pipeline. Finally, the results returned from the server are merged for rendering. Further, we describe the complete operational flow of the AoDNN as follows:

STEP. 1. In the offline phase, AoDNN collects and analyzes layers' inference performance with DNN Profiler, including compressed DNNs executed on various devices and platforms.

STEP. 2. DNN Partitioner establishes a multi-objective automatic offloading optimizer. It satisfies application constraints such as latency and mobile energy based on DNN Profiler's prediction framework. Then, it takes the current contexts as the input to generate manifest files that can be deployed and configured for the web services. Unlike conventional web offloading methods requiring developers to deeply analyze tasks and package them to generate offloading manifest files, this step is fully automated and requires no involvement.

STEP. 3. Once the auto-offloading manifest files are generated, the web service providers simply deploy these standard files to the web server and related web services.

STEP. 4. In the online phase, the browser initiates the DNN inference request to the web server in the cloud (including the edge cloud), which is the same as an ordinary web request.

STEP. 5. The web server returns to the browser the files such as HTML, CSS, and JavaScript, including the manifest files for auto-offloading generated in the previous step. They are routinely used for rendering pages for web services. Similarly, this step is consistent with the normal web service process.

STEP. 6. This step uses multiple web workers to enable seamless DNN offloading. Before loading the DNN model and executing inference, the browser first loads the context monitoring function from "auto-offloading.js" and determines whether the offloading is triggered based on "*MakeDecision()*". If the current DNN inference can be made natively, the corresponding DNN model is directly loaded on the browser for execution. Otherwise, the "*DynamicPartitioner()*" provides the best partitioning and offloading the snapshot.

STEP. 7. When the edge server receives the snapshot from the browser, the Service Context Manager (SCM) restores the inference snapshot based on the DNN model files. Then, SCM computes the graph synchronized from the web server side, awakes the inference workers, and returns the results. The process shown in Fig. 2 does not block the main thread of the browser by using sub-threads and web workers to execute the inference. Hence, it makes the inference and offloading process transparent to the users.

B. DNN Profiler

This section describes how DNN Profiler provides AoDNN with a learning-based DNN layer inference prediction framework in Fig. 2. It provides a polynomial regression learner

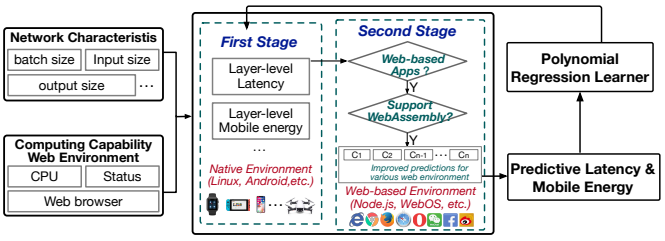


Fig. 2. Platform-aware predictive framework to support offloading decisions.

to search for the optimal coefficients considering the layer-inference prediction and memory access for the web-based environment run on devices with different computing capabilities. We first employ a similar definition of the layer-level prediction model in [23] and consider equally essential factors including running environment and the computing capability of the used device, representing the inference cost $C_l(\mathbf{x})$ of a given DNN layer l can be described as:

$$C_l(\mathbf{x}) = \sum_j \alpha_j \cdot \prod_{i=1}^{D} x_i^{e_{ij}} + \sum_k \beta_k \cdot \mathcal{F}_k(\mathbf{x}), \qquad (1)$$

where $\mathbf{x} \in \mathbb{R}^D$ is the feature vector, dimension D includes the input size, the output size, the batch size, the filter size, the kernel number and shape, the stride size, and the padding size, current computing capability and state of the used device. Note that the dimensions of different types of DNN layers may be different from each other. For instance, the kernel, the stride size, and the padding size is only used in convolutional layers. $e_{ij} \in \mathbb{N}$ is the exponent of x_i in the j^{th} polynomial term, and $\forall\, j, \prod_{i=1}^{D} x_i^{e_{ij}} \leq K$ where K is the regular degree of polynomial. Besides, the memory access cost and the floating-point operations of the DNN layer can also not be ignored,

thus using particular polynomial term $\mathcal{F}(\mathbf{x})$ to denote these two critical features. α and β are the coefficients to learn.

We can learn the coefficients to fit the DNN layer prediction models of the latency and mobile energy by inputting DNN layer features and the used device information based on such a prediction model. Although the above prediction model considers the possible factors of DNN characteristics, especially the computing capability and running state of the used device, the prediction model is aimed at executing the DNN directly on the mobile devices apps. Thus, it cannot be directly used for various web-based apps due to the performance degradation and differences among various browsers such as Chrome, Safari, Opera, and embedded app-browsers. Besides, whether or not to support WebAssembly technology also affects the predictions. To eliminate the prediction error caused by various browsers and using WebAssembly as much as possible, we use the first-order polynomial $B_u(c)$ to fit the DNN layer predictions running on different browsers without WebAssembly. We also use the quadratic polynomial $B_s(c)$ to fit the DNN layer predictions that support WebAssembly.

$$B_u^i(c) = \gamma_i \cdot c + \epsilon_i, \tag{2}$$

$$B_s^i(c) = \gamma_i \cdot c^2 + \delta_i \cdot c + \epsilon_i. \tag{3}$$

Where $c \in C_l(x)$ denotes the prediction result of a given DNN layer l using Eq. (1), γ and δ are the coefficients and ϵ is the constant term. As shown in Fig. 2, we can provide different prediction models to adapt different browsers that support or do not support WebAssembly, thus laying an essential foundation for the partitioning decision in AoDNN. Also, to learn the coefficients of proposed models, we first search the optimal order of polynomial of various predictions, and using ElasticNet [24] to select the best model of the polynomial model with the lowest cross-validation Mean-Square-Erro (MSE):

$$\mathcal{J}(\theta) = MSE(C, \hat{C}; \theta) + \mu_1 \cdot \sum_{j=1}^{n} \|\theta_j\| + \mu_2 \cdot \sum_{j=1}^{n} \|\theta_j\|_2^2 . \tag{4}$$

Based on the definitions mentioned above, we collect the actual inference latency and mobile energy of common DNNs in various devices with different CPU frequencies and collect latency and mobile energy when executing common browsers and embedded browsers. Our training data includes 928 convolutional layers, 220 pooling layers, and 156 fully connected layers collected from common DNN networks such as AlexNet, VGG16, MobileNet, ShuffleNet, ResNet, etc. Since the dropout layer is an effective means of preventing overfitting during the DNN model training phase, it can be ignored in the inference phase. The configurable parameters of the convolution, pooling, and fully connected layers take the input data, output data and other features as the variables. ReLU and Normalization layers are handled similarly. Before we generate training and testing data for fitting prediction models of the DNN layers on different platforms, including the edge server, various mobile devices, and web browsers, we introduce tools and methodologies to measure the basic profiles. First, the total latency can be calculated based on two timestamps before and after the DNN target execution. We repeat the same DNN inference multiple times and use the average latency to reduce random errors. Then, we use a hardware power monitor of AAA10F [25] to provide a stable voltage of 3.7V for devices and obtain the system energy cost, such as the screen brightness cost in the standby state. Last, we use a command-line tool that allows the user to communicate with an emulator instance to control the CPU frequency of the mobile device by Android Debug Bridge (ADB) [26].

C. Dynamic Partitioner

In this section, we explain how to provide dynamic partitioning facing various contexts. Supposing that there is only one partitioning point, then the DNN model is partitioned into two parts for executing DNN inference on mobile web and the edge, respectively. Note that the mobile web is inclined to execute the front part of the DNN inference because executing the front part of the DNN means that the mobile web has transmitted the input to the edge. The optimal solution at this time is executing the whole inference, which degenerates to the edge-only approach.

We give basic notations to describe better the DNN partitioning mentioned above as follows. λ_i, ψ_i, and w_i are the input data size, output data size, and the weight size of i^{th} layer, respectively. tw_i and te_i are the i^{th} layer latency on the mobile web and the edge, respectively. tu_k is the k^{th} layer transmitting latency. ew_k is the energy consumption of k^{th} layer inference. TM and TW are the model loading latency and inference latency on the mobile web, respectively. TU and TE denotes the uploading latency and inference latency on edge, respectively. EM and EW are the mobile energy of the loading model and inference on the mobile web. EU is the energy consumption to upload data to the edge. Besides, dc means the computing capability of the mobile device. Specially, we use T_i to denote the intermediate result of partial inference on the edge or the mobile web, $r(T_i)$ and P_i represent the transmission rate and mobile energy, respectively. Then, we can formulate the partition-offloading problem as a multi-objective optimization with two goals, including latency and mobile energy, which can be defined as

$$\begin{aligned} \mathcal{P}_1: \ & min \ \ F(k) = (T_{total}(k), E_{mobile}(k)), \\ & s.t. \ \ T_{total}(k) < T_{ubound}, \\ & \quad E_{mobile}(k) < E_{ubound}, \\ & \quad k \in \{0, 1, ..., N\}. \end{aligned} \tag{5}$$

Where $T_{total}(k)$ and $E_{mobile}(k)$ denote optimization objectives of the overall inference latency and mobile energy consumption. The units of the two objectives are ms and J, respectively. $k \in \{0, 1, ..., N\}$ guarantees the partitioning is valid. The two conditions constraint the upper bounds of inference latency and mobile energy. $T_{total}(k)$ is the overall processing latency of executing inference between the mobile web and the edge server, consisting of communication latency and inference latency. Because the partitioning point leads to different latencies for the mobile web and the edge server,

we search the optimal partition k to balance the latency and mobile energy.

$$\begin{aligned} T_{total}(k) &= T_{inference} + T_{communication} \\ &= TW_k + TE_k + TM_k + TU_k \\ &= \sum_{i=0}^{k} tw_i + \sum_{i=k+1}^{N} te_i \\ &+ \sum_{i=0}^{k} (w_i / r(w_i) + \lambda_{i+1} / r(\lambda_{i+1})). \end{aligned} \tag{6}$$

Similarly, mobile energy consumption mainly consists of communication and inference. Different partitioning points may cause various amounts of computation for the mobile web and the edge server, which means that the mobile web may consume different mobile energy levels. Besides, the edge server's output results are assumed to be of small size, and thus the feedback latency can be ignored.

$$\begin{aligned} E_{mobile}(k) &= E_{inference} + E_{communication} \\ &= EW_k + EM_k + EU_k \\ &= \sum_{i=0}^{k} ew_i + EM_i + EU_i \end{aligned} \tag{7}$$

To solve multi-objective optimization in linear complexity, which is suitable for an efficient scheduler on the mobile web, such online offloading scheduling problem $\mathcal{P}_1$ can use the linear weighting method to formulate it as

$$\begin{aligned} \mathcal{P}_2: \ & min \ \ T_{total}(k) + \eta \cdot E_{mobile}(k), \\ & s.t. \ \ T_{total}(k) < T_{ubound}, \\ & \quad E_{mobile}(k) < E_{ubound}, \\ & \quad k \in \{0, 1, ..., N\}. \end{aligned} \tag{8}$$

Since the objective function is the weighted sum of the execution latency and mobile energy with (in ms·J^{-1}) the weighting factor η, we set η as the ratio of T_{ubound} to E_{ubound} to adjust the tradeoff between the latency and mobile energy. Generally, service providers set the T_{ubound} and E_{ubound} to control the η for balancing the latency and mobile energy according to the application characteristics. We use a traditional but suitable and effective solution to solve the proposed multi-objective optimization, which is the weighted sum of latency and mobile energy for the following reasons: *(i)* The goal of establishing such multi-objective optimization is to balance the execution latency and mobile energy to avoid falling into the partition with one goal of latency or mobile energy. The weighted sum of latency and mobile energy can efficiently solve the problem for AoDNN, and our experimental results also show the effectiveness. *(ii)* The more important reason is that complex and accurate solutions for multi-objective problems such as genetic algorithms and reinforcement learning algorithms require much time and resources to search for the optimal solution. Thus, extra computation latency is unacceptable when executing complex DNNs. Although such a traditional solution is simple, it is suitable for web platforms and can efficiently execute with little latency and energy consumption than other solutions.

D. Context-Driven Offloading and Runtime Execution

This section describes how AoDNN uses the generated "auto-offloading.js" to offload the inference between the mobile web and the edge server.

1) Preparing before Offloading: Web browser loads the service files and auto-offloading file and regularly imports them, using <src="auto-offloading.js"> to load the service. Web browser first establishes the communication pipeline with the interface of the edge server, thus avoiding the time waste of establishing communication links when triggering the offloading. Hence, the offloading prerequisites needed for the runtime phase is pre-prepared at the initial service loading before executing the DNN inference.

2) Context Sensing and Runtime Offloading: When the browser initializes and loads the "auto-offloading.js" file, the offloading manager first detects the context through the monitoring worker as input to awake the decision worker. The decision worker obtains the optimal DNN partitioning according to the DNN partitioner and determines whether to trigger offloading. In Fig. 3, once the partition point is at the last layer, all inferences are performed locally, otherwise triggering offloading in all other cases. When the offloading is triggered, offloading manager generates a snapshot and transfers it to the edge server. The generated task snapshots do not contain the model data required for DNN inference to reduce network bandwidth consumption. In addition, when the decision worker provides a defined offloading scheme, the offloading worker simultaneously requests from the edge server to download the DNN model data required for inference. Note that AoDNN only loads the necessary service files from the web server for initialization, and the edge server provides the DNN models with high bandwidth consumption.

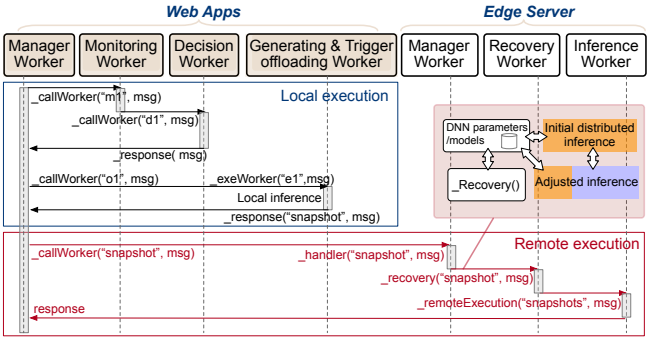


Fig. 3. Seamless execution diagram among various web workers.

Next, we describe how the offloading worker generates a snapshot that can be quickly recovered in the edge server. In general, extracting and saving the execution state of the web task as another application in the form of source code is known as the snaptshot [27], [28]. It greatly facilitates us to quickly recover the computational tasks saved by the snapshot in the edge server. We describe the flow of the snapshot-based offloading in Fig. 4. Assuming a DNN inference $model\_V = inf()$ of the web browser, the decision worker triggers the offloading, generates a snapshot of the current application context, and sends it to the edge server. Then, the edge server resumes and runs the DNN inference contained in the snapshot, and after executing $inf()$, it packages the current computation state, result, and context into a

new snapshot and sends it to the browser. The web browser receives the snapshot and resumes to immediately execute the subsequent DNN inference. When snapshot offloading is triggered, neither the thread nor the application waits for the response from the server. When a snapshot is received from the browser, the service context manageer (SCM) forwards it to the headless browser without GUI, whose processes such as UI rendering are streamlined to a pure computational version, thus effectively enhancing the recovery and operation of the snapshot task. Note that when the communication pipeline between the browser and the edge server is abnormal, the current snapshot is discarded, and the browser needs to re-initiate a new request.

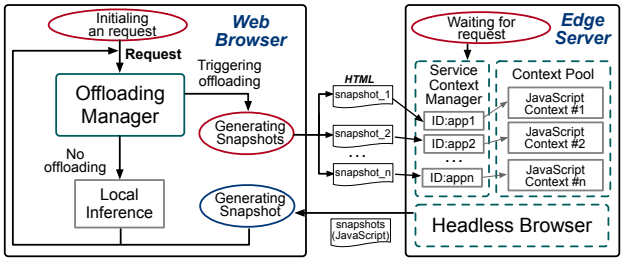


Fig. 4. Offloading context management and sychronization.

3) Snapshot Recovery and Remote Execution: We have described how the web browser triggers and generates DNN offloading snapshots and resumes the snapshots. This subsection focuses on how the edge server can then synchronize different offloading snapshots and provide remote real-time execution. SCM is responsible for performing offloading snapshots from the web browser and forwarding them to the corresponding inference worker for execution. Since SCM needs to process snapshots initiated by different web users and different web applications, the WebKit engine on the edge server needs to support the independent operation of snapshots with different contexts. As shown in Fig. 4, the SCM manages snapshots using *<key:value>* pairs. We use different identification numbers to distinguish the corresponding contexts of the snapshots sent from the web to the edge server. When an abnormal occurs, the edge server clears these expired contexts, and the browser will refresh, load, and hang the offloading services again.

III. Evaluation

This section introduces the methodology and settings for evaluation, including benchmarks, datasets, and detailed settings. Second, we focus on the effectiveness of the DNN profiler and partitioner in AoDNN compared with the benchmarks. We also compare AoDNN with the common web offloading methods to illustrate the advantages of AoDNN. Last, we deeply analyze the parameters involved.

A. Methodology and Settings

1) Benchmarks and Datasets: We compare AoDNN with representative DNN partition-offloading approaches such as Neurosurgeon [12], JointDNN [13], and MAUI [29], a control-centric method makes decisions about regions of code. We use Neuro-L and Neuro-E to denote optimal objects of the overall latency and the mobile energy, respectively, in Neurosurgeon. We also use JointDNN-B to denote the status with limited battery constraints. The datasets are the CIFAR-10 and ImageNet, widely used datasets for image recognition.

2) Mobile device and edge server setup: We used a HUAWEI Mate9 smartphone running the Android system with the Firefox browser, which is equipped with a CPU of eight cores (four cores of 2.4 GHz and four cores of 1.8 GHz) and 4 GB RAM. For the edge server, we used a regular server running Ubuntu 18.04 LTS. We set $T_{ubound} = 2.5$ and $E_{ubound} = 9$ for MobileNet and $T_{ubound} = 1.2$ and $E_{ubound} = 24$ for AlexNet. E_{ubound} can be calculated by $E_{ubound} = C \cdot U \cdot \xi \cdot 3600$. C denotes battery capability of the smartphone. U denotes the voltage. ξ is the percentage of used battery.

3) Communication setup: We describe the core network topology with an average downlink bandwidth of 150 Mbps and an average uplink bandwidth of 40 Mbps. We use Wonder Shaper [30] to limit the bandwidth of network adapters and simulate a variety of communication conditions such as 3G, 4G, and WiFi. Communication energy consumption of uploading and downloading depends on the network throughput (B_u and B_d). As indicated in [31], uplink and downlink energy consumption can be modeled with a linear equation, which is relatively accurate with less than 6% error rate. We show the average download and upload speed of a network of different network statuses in Table I, including the equation's parameters for the network condition.

TABLE I
Trasmission Power Modeling.

–	α_u(mW/Mbps)	α_d (mW/Mbps)	β_p(mW)
3G (1∼3Mbps)	868.98	122.12	817.88
4G (5∼12Mbps)	438.39	51.97	1288.04
WiFi (30∼100Mbps)	283.17	137.01	132.86

$$P_u = \alpha_u \cdot B_u + \beta_p$$
$$P_d = \alpha_d \cdot B_d + \beta_p \quad (9)$$

B. Effectiveness of DNN Profiler and Partitioner of AoDNN

1) Latency performance: We present the performance of AoDNN in various network conditions, which fixes the CPU frequency with four 2.4 GHz cores in Fig. 5. Besides, Fig. 5(a) and Fig. 5(b) describe the latency performance of four partition-offloading approaches for executing AlexNet over CIFAR-10 and ImageNet (the model sizes are 90.9 MB and 249 MB, respectively). Fig. 5(c) and Fig. 5(d) show the latency results of four approaches for executing MobileNet over CIFAR-10 and ImageNet (model sizes are 3.4 MB and 14.3 MB, respectively). In particular, AoDNN improves overall latency by 1.8x on average and up to 20.4x on MobileNet.

The main results are: (1) When the network bandwidth reaches 3 Mbps from 1 Mbps, AoDNN, JointDNN-B, and Neuro-L will offload part of the DNN computations to the mobile web, requiring much time to load and execute DNN layers. However, MAUI provides the fixed partitioning point

for each network and dataset because its offloading decision is the code (functions). Thus, MAUI still loads large models at

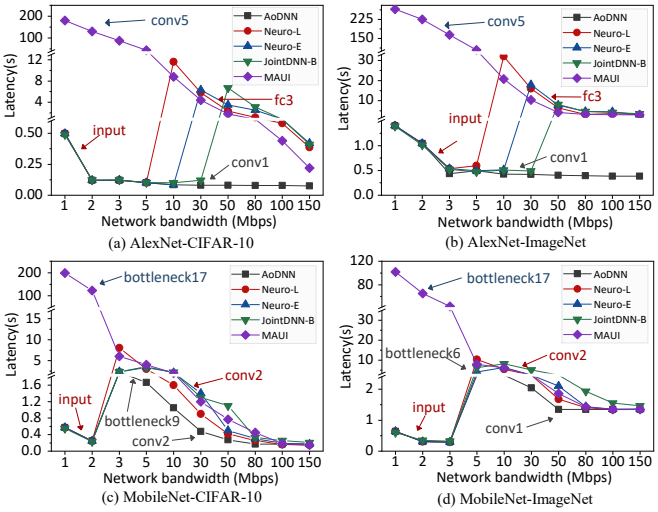


Fig. 5. Latency performance when controlling the network bandwidth.

low network speed. As network bandwidth increases, MAUI has lower latency than Neuro-L except on MobileNet of CIFAR-10. MAUI is inclined to offload more computation to the edge server, which means smaller loading latency. (2) Neuro-L's latency dramatically rises when the partitioning point changes from the input to the middle layer. Neuro-L has no partition at low speed and can be viewed as the edge-only approach. As network bandwidth increases, Neuro-L's partitioning point is changed to the fully connected layer of AlexNet and conv2 layer of MobileNet, resulting in large model loading latency. (3) When faced with a large DNN models or weak network speed, we find that AoDNN can avoid large model loading latency by changing to the edge-only scheme.

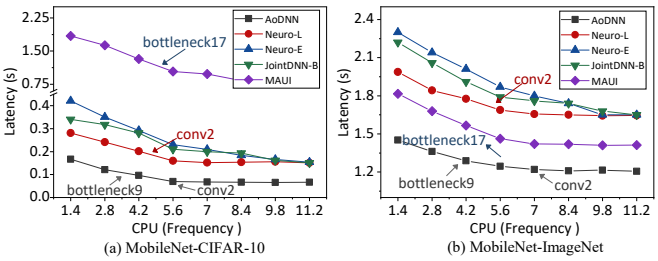


Fig. 6. Latency performance when controlling the computing capability.

We present the layer performance of AoDNN and other offloading approaches of the mobile device's various computing capabilities in Fig. 6. AoDNN can adjust the partitioning point and reduce the inference latency by perceiving the mobile device's computing capability. For example, AoDNN's latency is significantly lower than JointDNN-B and Neuro-L when accumulating CPU frequency is lower than 5.6 GHz. Neuro-L's partitioning point is at conv2 without adjustment, which means that the overall latency is not apparent, and the curve is flatter than others. When accumulating CPU frequency increases to 5.6 GHz, AoDNN's latency tends to be stable. This is mainly influenced by the mobile device's multicore computing architecture and other indicators, which cannot provide linear growth.

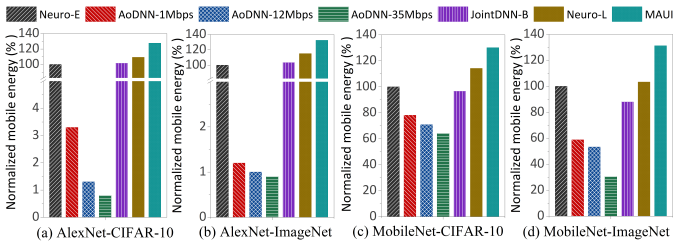
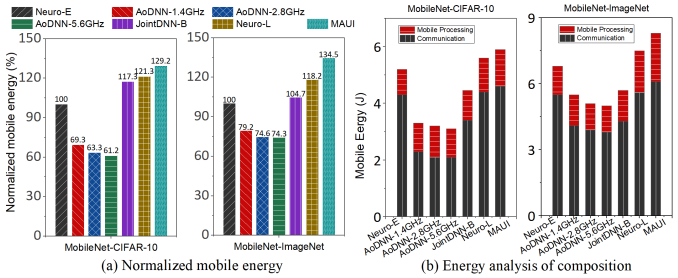


Fig. 7. Energy improvements of AoDNN in various network conditions.

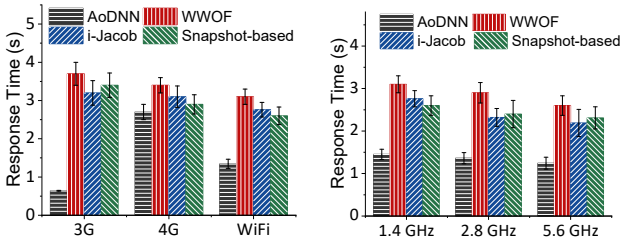
2) Mobile energy improvements: Fig. 7 shows the mobile energy consumption achieved by AoDNN in various network conditions, normalized to the status quo approach, which fixes the mobile device of CPU with four 2.4 GHz cores. AoDNN can reduce mobile energy consumption by 22.5% on average and up to 32.9% on MobileNet. (1) For AlexNet on CIFAR-10 and ImageNet, it can be seen that AoDNN reduces mobile energy consumption by 98% and 99% on average and up to 99.2%. This happens because AoDNN considers large amounts of model loading latency and provides the partition decision at the input layer to avoid large energy consumption on data transmission. Thus, AoDNN can be viewed as the edge-only approach with lower mobile energy consumption on transmitting image tasks rather than loading partial DNN model and inference locally. However, the status quo approaches calculate the mobile energy of transmitting intermediate results while ignoring loading model latency. (2) For MobileNet on CIFAR-10 and ImageNet, AoDNN reduces mobile energy consumption by 22.5% and 49.1% on average and up to 32.9% and 69.5%. The model size of MobileNet on CIFAR-10 is 3.4 MB, which is suitable for the mobile web. With the increase of network bandwidth, AoDNN provides dynamic partition while Neuro-E continuously partitions MobileNet at the layer of bottleneck9 and bottleneck6, respectively. For example, when the network bandwidth is 35 Mbps, AoDNN partitions MobileNet at the layer of bottlenect14 with the loading model size of 1.8 MB, which acquires energy improvement of 29.2%. Besides, with the increase of network bandwidth to 35 Mbps, AoDNN changes partitioning points at the layer of conv2, which consumes similar mobile energy to Neuro-E. In conclusion, AoDNN's partition is more flexible to dynamic network bandwidth, DNN model, and different datasets.

Fig. 8 shows that AoDNN reduces mobile energy consumption up to 38.8% and 25.7% for CIFAR-10 and ImageNet, as the CPU frequency of the mobile device increases. However, Neuro-E partitions MobileNet at the constant layer of fc1, which ignores the variations of mobile devices. For example, when we compare the mobile energy consumption between 1.4 GHz mobile device and 2.8 GHz mobile device, their mobile energy consumption differs at the same partitioning point. AoDNN adapts its partitioning point by perceiving the mobile device's computing capability and estimating the



(a) Normalized mobile energy (b) Energy analysis of composition

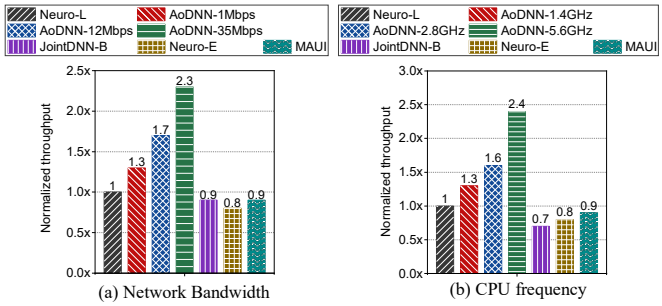
Fig. 8. Energy consumption with various compuitng capabilities.



(a) Various network bandwidths (b) Various CPU frequencies

Fig. 10. Comparisons with common web offloading approaches.

inference latency and mobile energy. For example, when the CPU frequency of the mobile device is 1.4 GHz, which means weak computing capability, AoDNN offloads a small portion of computations to the mobile web browser and avoids the loading latency and mobile energy of the partial MobileNet model. When the CPU frequency increases to 5.6 GHz, AoDNN adjusts the partitioning point to fc1 layer, which increases the amount of computation on the mobile web.



(a) Network Bandwidth (b) CPU frequency

Fig. 9. The edge server improvement of AoDNN in various contexts.

3) Throughput improvements: We evaluate AoDNN's throughput of the edge server normalized to the status quo approach using BigHouse [32] in Fig. 9. AoDNN can improve the throughput of the edge server by 1.2x on average and up to 2.4x. When the mobile users are connected to the edge server with a network bandwidth of 35 Mbps, AoDNN achieves 2.3x throughput improvement. As wireless connection changes to 12 Mbps and 1 Mbps, the throughput improvements become more significant 1.3x for 1 Mbps and 1.7x for 12 Mbps. AoDNN adapts its partition decision and pushes large portions of the MobileNet computation to the mobile web browser as the wireless connection quality becomes ideal. Also, we present the AoDNN's throughput improvement as the increase of CPU frequency of the mobile device in Fig. 9(b).

C. Comparison with Common Web Offloading

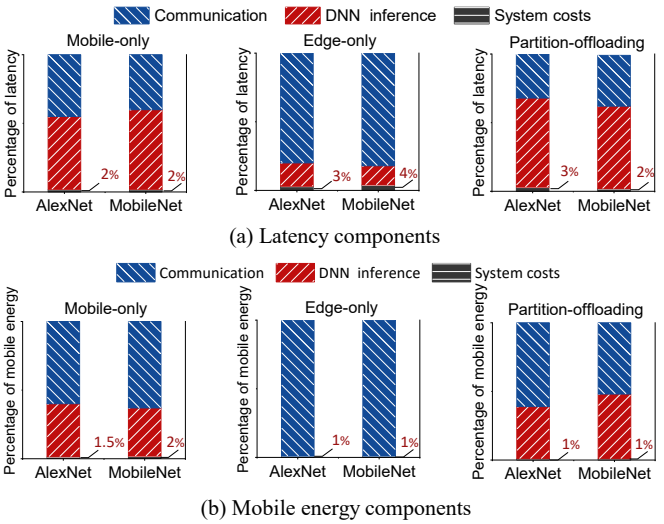
To verify the advantages of AoDNN over the existing web offloading approach for DNN inference tasks, we compare the response time of AoDNN by executing the MobileNet-ImageNet model with the advanced web offloading approach proposed for standard JavaScript tasks, including WWOF [8] and i-Jacob [11]. Besides, we also analyze AoDNN against a snapshot-based approach for web machine learning tasks [28].

The results in Fig. 10 show that: (i) AoDNN has a significant advantage over the other two common web offloading methods for DNN inference tasks. The snapshot-based approach also exhibits a lower response time than WWOF and i-Jacob when the network bandwidth is weak. This indicates that the conventional offloading mechanism is an inherent disadvantage for such a special computational task. (ii) We see that when the network bandwidth switches to WiFi, AoDNN shows a higher response time than the snapshot-based method because it offloads all of DNN inference to the edge server regardless of the network status. However, AoDNN offloads the part of DNN inference to the edge server according to the dynamic decision, and the inference worker mainly causes the increase in response time on the web performing partial inference. (iii) AoDNN outperforms all other methods in the high computing capability when in a fixed WiFi network. It still performs better than WWOF and i-Jacob in a low computing capability, similar to snapshot-based methods and analyzed above. In summary, we conclude that DNN inference is a special class of computational tasks for which the conventional web offloading method is not applicable.

D. Analysis of AoDNN

1) Analysis of Eq. (5): Latency and energy costs are the most critical indicators for executing optimal DNN computation between the mobile web and edge server. However, DNN computation consists of DNN inference, communication, memory access, and other system costs. We present the latency components executing AlexNet and MobileNet on CIFAR-10 between the mobile web and the edge server and show the energy cost components of the same DNN execution of the mobile device in Fig. 11. We observe that data communication and DNN inference occupy almost latency and energy consumption for a DNN task. Simultaneously, memory access and other system cost only occupy a small portion, which has little impact on the partition decision. Besides, to the best of our knowledge, most of the existing offloading approaches, including DNN and non-DNN task offloading, mainly consider the latency and energy cost in terms of data communication and task computation, such as [12], [29], and [13]. Thus, it is reasonable to consider communication and DNN inference as the dominant indicators in AoDNN.

2) Analysis η of Eq. (8): To show the impacts of the parameter η, we conduct experiments of MobileNet on CIFAR-10 to illustrate the effects of η in Fig. 12, which presents



(a) Latency components

(b) Mobile energy components

Fig. 11. Componets analysis of latency and mobile energy.

dynamic partition layers of AoDNN with different η. The results show that AoDNN's partition can balance the goal of latency and energy consumption when η ranges from 0.3 to 0.7. Besides, when η changes too little or too large, AoDNN degenerates into a single-objective partition that only considers the latency or energy consumption (similar to Neurosurgeon). In this work, we set the η for AlexNet on CIFAR-10 and ImageNet as 0.24 and 0.37. As for MobileNet, we set the η as 0.45 and 0.53 for CIFAR-10 and ImageNet.

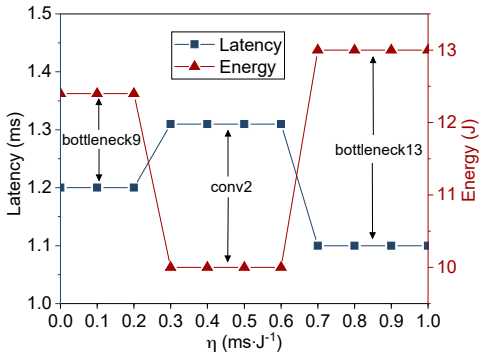


Fig. 12. Impacts of η on AoDNN's partition.

IV. Related Work

General offloading approaches. Existing frameworks make predictions about when and where to offload computation, and the correctness of the prediction dictates the final performance improvements for the mobile application. COMET [33] mainly aims to optimize the execution time by offloading a pre-defined thread and ignores the amount of data to transmit, network conditions, and others. Odessa [34] lacks consideration of the entire application during partition decisions and only considers the execution time and partial functions. CloneCloud [35] presents a flexible architecture for the seamless use of computation to augment mobile applications. It partitions applications automatically based on a combination of static analysis and dynamic profiling. MAUI's [29] offloading mechanism is better in that it makes predictions for each function invocation separately. It considers the entire application when choosing which code or function to offload. The most related offloading approach is common web offloading, such as WWOF [8], PIOS [9], and i-Jacob [11]. However, most of them aim at common JavaScript computations rather the DNN inference.

DNN inference on the web platform. We introduce some inference libraries for the web platform. JavaScript and WebAssembly are the dominant methods to execute deep learning inference on the mobile web browser. Typically, CaffeJS [36], Keras.js [37], TensorFlow.js [38], WebDNN [39] and ONNX.js [40] aim at executing DNNs on the same platform, rather than for distributed platforms. The most relevant techniques to this paper are aiming to offload computations of DNNs from the backend server to the end device [21]. Neurosurgeon [12] automatically chooses the partitioning point by pursuing the optimal latency or energy consumption to offload DNN computation. Edgent [16] searches the adaptive partition of DNN computation based on the Neurosurgeon, which aims to accelerate DNN inference through an early exit at a proper intermediate DNN layer. They also ignore the real-time loading of the partitioned model and lack the adaptive ability to face dynamic contexts, especially the computing capability of devices. Besides, they only consider single-objective optimization (either latency or energy consumption) in the partition phase [13]. Our work not only provides context-aware partition but also implements a distributed framework to execute DNN between the mobile web and the server.

V. Discussion and Conclusions

We discuss the contributions and the practicality of AoDNN for the mobile web. First, AoDNN implements an auto-offloading framework to migrate DNN inference from the mobile web to the edge server smoothly, which lays a good foundation for ubiquitous web applications to broaden the mobile web's capabilities, relieve the computing pressure, and improve the throughput of the edge server. Second, in this work, we mainly implement AoDNN on image classification as examples. Since the core contribution of AoDNN is to provide an adaptive DNN execution scheme between the mobile web and the edge server for ubiquitous web applications, it can be applied to other deep learning fields with similar models.

We conclude that this work proposes a seamless and auto-offloading approach to execute elastic DNN inference between the mobile web and the edge server. We analyze the contextual factors on different datasets and networks and define three critical contexts: the network condition and the mobile device's computing capability. Next, we explore the JavaScript computation offloading framework by proposing a snapshot-based framework. We propose an adaptive execution scheme of distributed neural networks for mobile web applications to adapt dynamic contexts. The experimental results demonstrate that AoDNN improves latency and mobile energy performance and is more suitable for web applications.

REFERENCES

[1] Y. Ma, D. Xiang, S. Zheng, D. Tian, and X. Liu, "Moving deep learning into web browser: How far can we go?" in *The World Wide Web Conference (WWW)*, 2019, pp. 1234–1244.

[2] X. Qiao, P. Ren, S. Dustdar, L. Liu, H. Ma, and J. Chen, "Web AR: A promising future for mobile augmented reality—State of the art, challenges, and insights," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 651–666, 2019.

[3] "Web of Things (WoT)," 2021, [Online]. Available: https://www.w3.org/WoT/.

[4] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with WProf," in *the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 473–485.

[5] X. S. Wang, H. Shen, and D. Wetherall, "Accelerating the mobile web with selective offloading," in *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, 2013, pp. 45–50.

[6] A. Gallidabino and C. Pautasso, "Decentralized computation offloading on the edge with liquid webworkers," in *International Conference on Web Engineering (ICWE)*. Springer, 2018, pp. 145–161.

[7] H.-J. Jeong, C. H. Shin, K. Y. Shin, H.-J. Lee, and S.-M. Moon, "Seamless offloading of web app computations from mobile device to edge clouds via html5 web worker migration," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2019, pp. 38–49.

[8] X. Gong, W. Liu, J. Zhang, H. Xu, W. Zhao, and C. Liu, "Wwof: an energy efficient offloading framework for mobile webpage," in *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2016, pp. 160–169.

[9] S. Park, Q. Chen, and H. Y. Yeom, "PIOS: A platform-independent offloading system for a mobile web environment," in *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*. IEEE, 2013, pp. 137–142.

[10] M. Yu, G. Huang, X. Wang, Y. Zhang, and X. Chen, "Javascript offloading for web applications in mobile-cloud computing," in *2015 IEEE International Conference on Mobile Services*. IEEE, 2015, pp. 269–276.

[11] X. Liu, M. Yu, Y. Ma, G. Huang, H. Mei, and Y. Liu, "i-Jacob: An internetware-oriented approach to optimizing computation-intensive mobile web browsing," *ACM Transactions on Internet Technology (TOIT)*, vol. 18, no. 2, pp. 1–23, 2018.

[12] 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*, vol. 45, no. 1, pp. 615–629, 2017.

[13] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN:An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, early access, Oct. 16,2019, doi: 10.1109/TMC.2019.2947893.

[14] S. Zhang, Y. Li, X. Liu, S. Guo, W. Wang, J. Wang, B. Ding, and D. Wu, "Towards real-time cooperative deep inference over the cloud and edge end devices," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (UbiComp)*, vol. 4, no. 2, pp. 1–24, 2020.

[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 (MobiCom)*, 2020, pp. 1–15.

[16] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*. ACM, 2018, pp. 31–36.

[17] Y. Huang, X. Qiao, P. Ren, L. Liu, C. Pu, and J. Chen, "A lightweight collaborative recognition system with binary convolutional neural network for mobile web augmented reality," in *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1497–1506.

[18] Y. Huang, X. Qiao, J. Tang, P. Ren, L. Liu, C. Pu, and J. Chen, "DeepAdapter: A collaborative deep learning framework for the mobile web using context-aware network pruning," in *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2020, pp. 834–843.

[19] Y. Huang, X. Qiao, P. Ren, L. Liu, C. Pu, S. Dustdar, and J. Chen, "A lightweight collaborative deep neural network for the mobile web in edge cloud," *IEEE Transactions on Mobile Computing*, early access, Dec. 2020, doi: 10.1109/TMC.2020.3043051.

[20] Y. Huang, X. Qiao, J. Tang, P. Ren, L. Liu, C. Pu, and J.-L. Chen, "An integrated cloud-edge-device adaptive deep learning service for cross-platform web," *IEEE Transactions on Mobile Computing*, early access, Oct. 2021, doi: 10.1109/TMC.2021.3122279.

[21] P. Ren, X. Qiao, Y. Huang, L. Liu, S. Dustdar, and J. Chen, "Edge-assisted distributed dnn collaborative computing approach for mobile web augmented reality in 5g networks," *IEEE Network*, vol. 34, no. 2, pp. 254–261, 2020.

[22] "Using AWS Lambda with CloudFront Lambda@Edge." 2021, [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html.

[23] C. F. Rodrigues, G. Riley, and M. Luján, "Energy predictive models for convolutional neural networks on mobile platforms," *arXiv preprint arXiv:2004.05137*, 2020.

[24] R. Durbin, R. Szeliski, and A. Yuille, "An analysis of the elastic net approach to the traveling salesman problem," *Neural computation*, vol. 1, no. 3, pp. 348–358, 1989.

[25] "Monsoon solutions," 2021, [Online]. Available: https://www.msoon.com.

[26] "Android debug bridge," 2021, [Online]. Available: https://developer.android.com/studio/command-line/adb.

[27] H.-J. Jeong and S.-M. Moon, "Offloading of web application computations: A snapshot-based approach," in *International Conference on Embedded and Ubiquitous Computing*. IEEE, 2015, pp. 90–97.

[28] I. Jeong, H.-J. Jeong, and S.-M. Moon, "Work-in-progress: snapshot-based offloading for machine learning web app," in *2017 International Conference on Embedded Software (EMSOFT)*. IEEE, 2017, pp. 1–2.

[29] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[30] "Wonder shaper," 2021, [Online]. Available: https://github.com/magnific0/wondershaper.

[31] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4g lte networks," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 225–238.

[32] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2012, pp. 35–45.

[33] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 93–106.

[34] M.-R. 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*. ACM, 2011, pp. 43–56.

[35] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[36] "CaffeJS," 2021, [Online]. Available: https://github.com/chaosmail/caffejs.

[37] L. Chen, "Keras.js," 2021, [Online]. Available: https://github.com/transcranial/keras-js.

[38] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel *et al.*, "Tensorflow. js: Machine learning for the web and beyond," *arXiv preprint arXiv:1901.05350*, 2019.

[39] M. Hidaka, Y. Kikura, Y. Ushiku, and T. Harada, "WebDNN: Fastest DNN Execution Framework on Web Browser," in *the ACM on Multimedia Conference (MM)*. ACM, 2017, pp. 1213–1216.

[40] "ONNX.js," 2018, [Online]. Available: https://github.com/Microsoft/onnxjs.