2018 IEEE 38th International Conference on Distributed Computing Systems

Computation Offloading for Machine Learning Web Apps in the Edge Server Environment

Hyuk-Jin Jeong, InChang Jeong, Hyeon-Jae Lee, Soo-Mook Moon
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Korea
jinevening@snu.ac.kr, wing.jeong@gmail.com, thlhjq@snu.ac.kr, smoon@snu.ac.kr

***Abstract*— Machine leaning apps require heavy computations, especially with the use of the deep neural network (DNN), so an embedded device with limited hardware cannot run the apps by itself. One solution for this problem is to *offload* DNN computations from the client to a nearby *edge* server. Existing approaches to DNN offloading with edge servers either specialize the edge server for fixed, specific apps, or customize the edge server for diverse apps, yet after migrating a large VM image that contains the client's back-end software system. In this paper, we propose a new and simple approach to offload DNN computations in the context of *web apps*. We migrate the current execution state of a web app from the client to the edge server just before executing a DNN computation, so that the edge server can execute the DNN computation with its powerful hardware. Then, we migrate the new execution state from the edge server to the client so that the client can continue to execute the app. We can save the execution state of the web app in the form of another web app called the *snapshot*, which immensely simplifies saving and restoring the execution state with a small overhead. We can offload any DNN app to any generic edge server, equipped with a browser and our offloading system. We address some issues related to offloading DNN apps such as how to send the DNN model and how to improve the privacy of user data. We also discuss how to install our offloading system on the edge server on demand. Our experiment with real DNN-based web apps shows that snapshot-based offloading achieves a promising performance result, comparable to running the app entirely on the server.**

Keywords—edge computing, cloud computing, computation offloading, web application, machine learning, neural network

I. INTRODUCTION

Machine learning (ML) requires heavy computations, especially for training and inference based on deep neural networks (DNN). This raises an issue of how to perform machine learning on *embedded devices* with scarce resources such as wearables or IoT devices. One promising solution is employing *edge computing* [1], which leverages decentralized computing infrastructures [2], such as fog nodes [3] or cloudlets [4], for *computation offloading*. Unlike centralized, datacenter-based cloud computing, edge computing allows the clients to use a closer computing server located at the edge of the network, e.g., Wi-Fi hotspots or a cellular station, with a lower network latency. So, the embedded clients can accelerate ML app execution by offloading the ML computations such as inference/training of DNN, to any nearby edge server.

Recent offloading techniques to use the edge servers for ML treat the edge servers either as *specialized* servers for specific applications or as *generic* servers for diverse applications. A typical example of the *specialized* servers is a computation server for video processing. A client device sends a streaming video input to the server, and the server performs an ML algorithm on the input for a specific application (e.g., traffic surveillance [5], video streaming [6], virtual reality [7]) and returns the result back to the client. On the other hand, *generic* edge servers can perform any computation that the client wants to compute, on demand. For this, the client first sends ML service software to an edge server to customize it, then the customized edge server handles the ML requests from the client. A well-known strategy of customizing the edge server is using a virtual machine (VM), such that the client migrates a VM image that encapsulates the ML software and its base system to the server, and installs it there for use thereafter [4] [14]. *Gabriel* is a recent work on the VM-based customization, which customizes the edge servers with cognitive engines for servicing mobile cognitive assistance [8].

In this paper, we propose a different approach to offload ML computations to generic edge servers. A programmer creates a regular, self-contained app for ML, runnable on the client device on its own. When a computation-intensive part (e.g., DNN inference) is executed, the *whole* app state is dynamically migrated to a nearby edge server. The app executes there continuously and returns back to the client with its updated execution state. We propose this approach for *web apps*, programmed using HTML, CSS, and JavaScript, whose portability advantage allows simpler implementation. Our idea is using the *snapshot* [9] [10] [11], which can save the current execution state of a web app in the form of *another web app*. If we run the snapshot on a browser, we can automatically restore the execution state and continue its execution from the point where it was saved. So, we take a snapshot just before executing a computation-intensive part on the client and send the snapshot to the edge server. The server performs the computation by simply running the snapshot on its browser, then takes a new snapshot which contains the computation result. The new snapshot is sent back to the client and run on the client's

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00154

IEEE computer society

browser to continue its execution. Consequently, we can offload ML computations to the edge server on demand.

Compared to the VM-based customization, snapshot-based offloading is much lighter since the snapshot merely contains the app execution state in the form of HTML/CSS/JavaScript code, not a VM image with the entire disk and memory state. Also, snapshot-based offloading still allows installing regular apps at the client, whereas the VM-based customization requires the client to carry the VM images for the back-end software as well. Although it has a language dependence unlike VM-based customization (i.e., it works only for web apps), snapshot-based offloading is highly portable since web apps are runnable on any devices and servers where a web browser is available. Finally, when we need to change the edge server during app execution (e.g., when a mobile client moves to a different service area), snapshot-based offloading can readily work on a new edge server since it has no dependence on the previous server, while VM-based customization needs to install the VM on the new edge server before using it.

Compared to the specialized ML offloading, the snapshot allows more flexible offloading since it can include any kind of computations as well as the ML algorithms; e.g., if the pre/post processing of the input/output data for ML is as heavy as the ML algorithms, they can also be offloaded to the edge server using the snapshot (either in the same snapshot with the ML algorithms or in a separate snapshot). We can even change the client's screen at the edge server if the execution of the snapshot includes the update of the DOM tree (which controls the screen display in the web app). This allows offloading more computations than ML-specialized edge servers, improving app performance further. Finally, snapshot-based offloading can reduce the burden of app programmers because programmers do not have to consider how to program the client and the server separately, how to send and receive the data, or how to coordinate execution between them; the snapshot will transparently reconcile the execution states between them.

We developed the snapshot-based offloading system for the DNN-based ML apps on the *WebKit* browser [12]. We propose optimizations to reduce the transmission time of the DNN model and to improve the privacy of user's data. We also discuss how to install our offloading system on the edge server on demand. In real web apps, snapshot-based offloading shows a promising performance result, comparable to running the app entirely on the server, even with the snapshot-related overhead.

The rest of the paper is organized as follows. We review DNN in Section 2. Section 3 describes the snapshot-based offloading for ML apps. Section 4 evaluates the proposed idea. The summary and future work are in Section 5.

II. BACKGROUND

This paper focuses on ML apps based on DNN, so we first review how DNN works before explaining how we used snapshot-based offloading for DNN apps.

A. Deep Neural Network

Neural network (NN) can be seen as a dataflow graph whose nodes are *layers*, each of which performs its operation on the input data and passes the output data to the next layer. The first layer of NN (*input* layer) receives the input data from a user and passes it to the next layer as the form of a vector. The next layer performs its vector operation and propagates the result to the next layer, until reaching the last layer (*output* layer). We call the series of layer execution as *forward* execution. Layers between the input layer and the output layer are called *hidden* layers. Typically, a NN with multiple hidden layers is referred to as a deep NN (DNN).

A vector operation of each layer has parameters, which are iteratively updated while the DNN is trained with training data (*training phase*). After the training phase is done, the parameters of each layer are fixed, so the DNN always performs the same operations (series of layer execution) on given input data. The DNN in this state can be used for inferring a result for a new input data (*inference phase*). In this paper, we focus on offloading the computation of the inference phase, because the training is typically performed in the powerful servers due to its resource requirements.

B. Convolutional Neural Network

Convolutional NN (CNN) is widely used in the field of image processing (e.g., image classification). We used CNNs for the experiments in this paper, so we briefly explain the characteristics of layers used in the CNN below.

Convolutional layer (conv layer): A conv layer performs a convolution function on input data by scanning the input data with its *filters* (n by n matrices). At first, the conv layer computes the weighted sum (convolution) of the upper left of the input data with one of its filters. Next, the filter slides by one pixel and computes the weighted sum of that position. The sliding of the filter continues until it covers the entire image pixels. Afterwards, the filter outputs a matrix (feature) by concatenating the result of each position. This process is repeated for every filter, and the output of conv layer is composed of the result of all filters. Typically, conv layers in modern CNNs have many filters [13] [15], so the output of a conv layer is prone to be larger than the input.

Max pooling layer (pool layer): A pool layer selects the maximum value among the values inside the window of the pool layer (n by n matrix). Like the conv layer, the pool layer slides the window until covering all input pixels and concatenates the result. Since values other than the maximum value are abandoned, the output of a pool layer becomes smaller than its input.

Fully connected layer (fc layer): An fc layer is composed of neurons each of which is connected to the outputs of the previous layer. Each neuron computes the weighted sum of its input using its parameters.

There are many other types of layers (e.g., normalization layer, softmax layer) used in the CNN, but explaining them exceeds the scope of this paper.

Fig. 1 shows the architecture and the data flow of GoogLeNet [13], illustrating how CNN classifies an image as one of 1,000 output labels in detail. A user gives an image

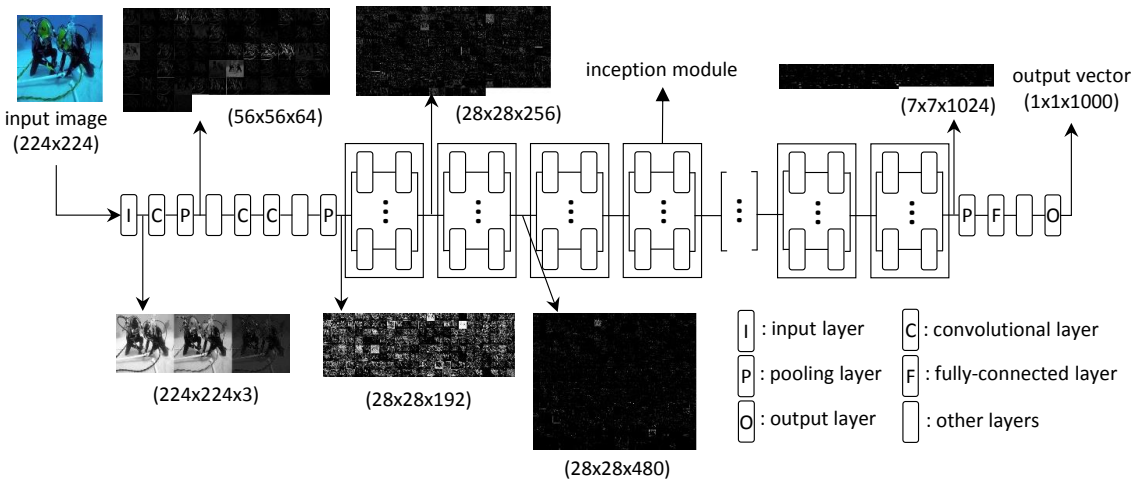

Figure 1: Architecture and intermediate computation result (feature data) of GoogLeNet

whose size is (224x224) to an input layer. After passing the input layer, the input image becomes a three-dimensional vector (224x224x3), composed of RGB values of each pixel. Next, conv layer and pool layer perform their operations on the pixel data, transforming the pixel data into the *feature data* whose dimension is (56x56x64), as shown on the Fig. 1. GoogLeNet introduced a special architecture, named inception module [13], which arranges multiple layers in parallel (depicted as squared boxes). *Inception* module extracts features from an input by using conv layers and pool layers arranged in parallel, and the features are concatenated into a single output vector and passed to the next layer. In the latter part of the network, there is an fc layer which transforms the extracted feature data into the output vector whose elements are the scores where the input image belongs to the output labels.

Fig. 1 also visualizes the output feature data of some DNN layers as a grayscale image (whose dimension is shown under the image). It is done by creating two-dimension images from the feature data and putting them together like tiles. For example, the feature data whose dimension is (56x56x64) in the upper left of Fig. 1 is visualized as 64 images, each of which is a 56x56 matrix. These visualized images show how the input data is transformed as it passes through the layers, which will be used for discussing the privacy of user data later.

III. Snapshot-based Offloading for ML Web Apps

This section introduces snapshot-based offloading proposed in [10] and describes how to exploit it for ML apps in the edge computing environment, with related issues and solutions.

A. Snapshot-based Offloading

We illustrate the snapshot-based offloading with an example ML web app in Fig. 2. The app loads an image when a user clicks the load button. Then, when the user clicks the inference button, the app inferences what the image is by using a pre-trained DNN model and shows the result on the screen. The sketch of the source code of the web app is also depicted. It is programmed using HTML/CSS/JavaScript. The browser will create a DOM-tree with the buttons and other components, when it parses the HTML code. The script tag will first load the DNN model used for image-recognition, and register an event handler for the click event for the inference button (the event handler for the load button is registered in the HTML code). The image-loading event handler will load the image and draw the DOM-tree on the screen. The inference event handler will execute the *inference()* computation on the model and add the result text to the DOM-tree to update the screen.

Snapshot-based offloading will proceed as follows. When the inference button is clicked and just before the time-consuming inference event handler is executed, the client takes a snapshot of the current execution state. The snapshot will contain the image to be analyzed, the functions of the app, and the code to restore the current execution state including the variables, the heap objects, and the DOM-tree. For example, if there is a global object *obj* with two properties *x* and *y* whose current values are 1 and 2,

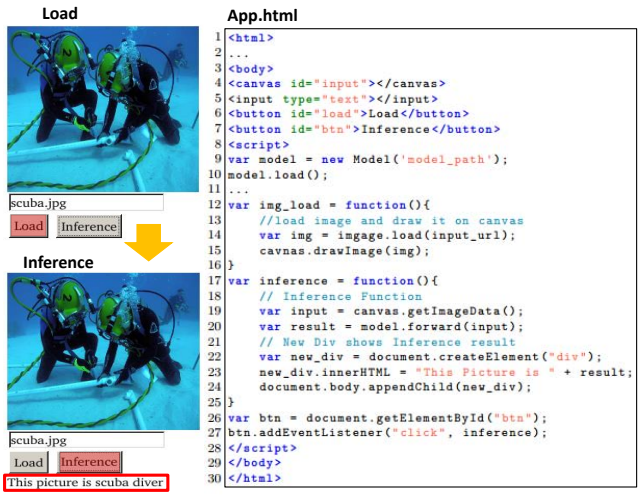

Figure 2: An example of ML Web App

Figure 3: Snapshot-based offloading for the example app

respectively, the snapshot will include *var obj={x:1, y:2};*. There is also the code to dispatch the event again at the server. We reduce the snapshot code size using various optimizations [10].

We transmit the snapshot to the server, and the server runs the snapshot on its own browser. Execution of the snapshot will first restore exactly the same execution state as when the client took a snapshot, and then continue the execution for the inference event handler. Then, the server takes another snapshot which contains a new execution state, now with the computation result added to the DOM-tree (it is actually JavaScript code to update the client execution state). The newly taken snapshot is sent back to the client, which runs the snapshot and continues the app execution. This is illustrated in Fig. 3 with a sketch of the two snapshots. By sending and receiving the snapshot between the client and the server, offloading can work seamlessly.

B. *Snapshot-based Offloading for ML Apps with Edge Server*

Snapshot-based offloading raises a few issues when applied to ML apps under the edge server environment. First, unlike normal apps, ML apps using DNNs have a large amount of app states mainly composed of the parameters of DNN models. For instance, one of our apps used for evaluation uses a DNN whose size is 44 MB. If we send the DNN model at runtime, it will take about 12 seconds for transmitting the model even under the good Wi-Fi network whose bandwidth is 30 Mbps (it will take longer under a worse network condition). Such a long network delay would affect feasibility. Secondly, the snapshot of ML apps may contain sensitive information about a user, so sending it to public edge servers may affect user privacy. For example, in the app described in Fig. 2, the input image can include private data (e.g., passport or social security number). When offloading occurs, the client will send a snapshot to the server which includes the image. The server can extract the input data by scrutinizing the snapshot, which is undesirable for the user. Finally, we expect our target edge servers are pre-installed with our offloading system in advance, but some edge servers might not be (e.g., due to using other offloading systems). In this case, it would be proper to install our system on the edge server dynamically at runtime.

To resolve these issues, we introduce pre-sending of DNN model, partial inference, and on-demand installation.

1) *Pre-sending Neural Network Model*

In our edge computing scenario, the client has a NN model trained in advance, and sends it to the server with the snapshot. The model has a much larger size compared to the snapshot itself, so if it is sent together with the snapshot, the transmission overhead would be non-trivial. So, we decide to send the model in advance to the server when the app starts, to reduce the data transmission overhead.

The pre-sending of an NN model works as follows. When a web app starts, the client device sends the NN model files (including the description/parameters of the NN) to the server. The list of the files which will be transmitted to the server are given by app developers. The server saves the files and sends an acknowledgement (ACK) message to the client. After receiving the ACK, the client just needs to send the snapshot without the model to the server when offloading occurs. This significantly reduces the transmission overhead (see Section 4.2). If offloading occurs before the ACK arrives, the client sends both the snapshot and the NN model, albeit it is slower.

2) *Partial Inference at the Client*

Generally, offloading ML apps require sending the user's private data to the edge server. Specialized servers for specific apps know in advance what kind of data they will receive, thus easy to collect the private data. Generic servers customized by VM would have less privacy concern, since although they will receive data from those apps using the back-end ML software in a VM image, the image is installed on demand. Generic servers for snapshot-based offloading can even reduce the concern since they can receive data from "any" apps, offloaded on demand by itself with no dependence on any software.

We introduce *partial inference* to allow the app developers to program their DNN apps in a more privacy-preserving way for offloading. The basic idea is illustrated in Fig. 4. We divide the DNN into two parts (front and rear) and execute the front part of the DNN at the client and the rear part at the server. The output of the front part of the DNN (feature data) is not easily recognizable by the human, as shown in Fig. 1. So, the client transmits the snapshot with the feature data, not the original input data, hiding the original data from the server.

To support partial inference in our framework, we

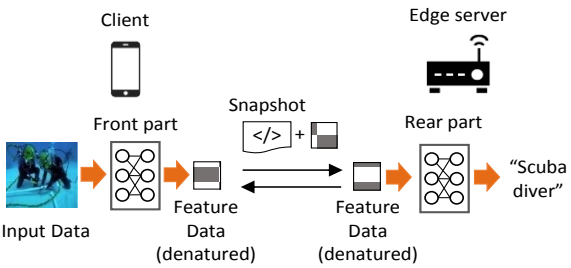

Figure 4: Partial inference for user privacy

```
1  <html>
2  ...
3  var feature;
4  var btn = document.getElementById("btn");
5
6  function front() {  // local execution
7    var image = canvas.getImageData();
8    feature = model.inference_front(image);
9    btn.dispatchEvent("front_complete");
10 }
11
12 function rear() {  // offloaded
13   var result = model.inference_rear(feature);
14   ...
15 }
16
17 btn.addEventListener("click", front);
18 btn.addEventListener("front_complete", rear);
19 ...
20 </html>
```

Figure 5: Example code of web app that uses partial inference

slightly changed the offloading process and the existing ML framework. Instead of using the *inference()* function (line 26 in Fig. 2), we produce two functions, *inference_front()* and *inference_rear()*, for inferencing in two steps, each of which executes the front part and the rear part of the DNN, respectively. The partitioning point of the front/rear part can be decided dynamically based on two factors. One is the execution time of each DNN layer, estimated by a prediction model for the DNN layers, as used in Neurosurgeon [16]. The other is the runtime network status. We estimate the total execution time for forward execution and select a partitioning point that can minimize the total execution time, while including at least one layer from the front part of the DNN to denature the input data.

Fig. 5 shows the web app code that uses partial inference in the example app in Fig. 2. The event handler inference is divided into two functions (*front* and *rear*), each of which is invoked in response to the "*click*" event and the "*front_complete*" event, respectively (line 17, 18). We will offload execution when the "*front_complete*" event is raised (not when the "*click*" event is raised). When a user clicks a button whose id is *btn*, a click event is raised, and *front()* will be invoked. *front()* executes the front part of the DNN by calling *inference_front* (line 8). After the partial inference, *front()* dispatches "*front_complete*" event (line 9), which is a custom event defined by the developer. Then, since we decided to offload execution when the "*front_complete*" event is raised, we capture a snapshot and transmit it to the server. The snapshot will reproduce the "*front_complete*" event at the server, and the server will continue the app execution, invoking *rear()* which executes the rest of DNN layers (line 13) and displays the result. After that, the server will capture a snapshot for the execution state with the result added and send it to the client to continue the app. In this scenario, the user's privacy is less violated, because the input data is not exposed to the server.

One issue for the effectiveness of partial inference is that if the server can reconstruct the original input data from the feature data, then it is useless. It is known that the input data can be reconstructed from the feature data by a *hill climbing* algorithm, given the information (layer types, parameters) of all layers that the input data passed thru to generate the feature data [17]. So, if the server can access the front part of the DNN model used for partial inference, the input data may be deduced from the feature data. We deal with this problem by pre-sending only the rear part of the DNN model. By not sending the front part of the DNN model, we can prevent the server from reconstructing the input from the feature data.

Partial inference may affect the performance of offloading, though. Since the client sends fewer layers of DNN (only the rear part of DNN) to the server, the data transmission overhead can be reduced, making the ACK arrive earlier. On the other hand, inference execution time will increase, because the front part of DNN is executed at the client with weak resources. We will evaluate the performance impact of partial inference.

3) Installing Offloading System at the Edge Server

Snapshot-based offloading requires the edge server to run our offloading server program for handling network connection, a web browser for executing the snapshot, and the support libraries. So, these programs and the libraries would better be installed on the edge server in advance. In fact, our target edge servers are likely to be equipped with some platform (e.g., a web platform like webOS or Tizen), so we can pre-install our offloading system there. If our system is not pre-installed (e.g., when the client meets a non-target edge server when it is on the move), we will need to install it on the edge server at runtime.

We can dynamically install our offloading system and the libraries on the edge server at once, by using a VM-based customization technique called *VM synthesis* [4]. The client sends *VM overlay*, which contains our offloading system (the server program, the libraries, and the web browser) to the edge server at runtime. The server can synthesize a VM instance from the VM overlay on the basis of the *base VM image*, which is a VM image that contains an OS necessary to run our offloading system. If the client sends its DNN model within the VM overlay, it would have the same effect of pre-sending the DNN model to the server. Once our system is installed on the edge server, we can offload any computation of any app to the server using the snapshot; this is different from the typical use scenario of VM-based customization, which customizes an edge server only for apps using their back-end software. We evaluate the installation overhead in Section 4.C.

IV. Evaluation

We evaluated if the proposed snapshot-based offloading works in real web apps. Our client board (Odroid-XU4) has an ARM big.LITTLE CPU (2.0GHz/1.5GHz Quadcores) with 2GB memory, while our server has an x86 CPU (3.4GHz Quadcore) with 16GB memory, connected by Ethernet. We limited the network bandwidth under 30 Mbps to emulate the network condition similar to Wi-Fi by using *netem* [18]. We used the WebKit browser version r150000. Our benchmarks are three image recognition apps with *GoogLeNet* [13], *AgeNet* [15], and *GenderNet* [15] models, implemented with *Caffe.js* ML framework [19]. *CaffeJS* loads a pre-trained NN model (trained by widely used NN framework named *Caffe* [20]) onto the web app and

performs forward execution using the model. We measured the execution time for the inference operation.

A. *Execution Time*

Fig. 6 shows the execution time for each benchmark when the app runs at the client only (*Client*), when the app runs at the server only (*Server*), and when the app runs both at the client and at the server with snapshot-based offloading (*Offloading*). There are three different configurations for *Offloading*: offloading before the ACK arrives at the client (i.e., no pre-sending), offloading after the ACK arrives at the client (i.e., pre-sending), offloading with partial inference after the ACK arrives (the partitioning point will be explained in Section 4.2). As expected, the server execution time is much shorter than the client execution time (since *Caffe.js* cannot exploit GPUs yet, the server execution time is much longer than it should be). Also, the offloading performance rapidly increases after the DNN model uploading is over. Most importantly, offloading after ACK shows an execution time similar to that of server's, even with the snapshot creation, restoration, and transmission overhead for both client-to-server and server-to-client migration. This implies that snapshot-based offloading can be a promising approach to offload DNN computations to generic edge servers.

There are two more things to note. One is that for AgeNet and GenderNet, offloading before ACK is even slower than the local client execution due to their large model size, so it would be better for the client to execute the DNN locally while the model is being uploaded to the server. The other is that partial inference is slower than full server-side inference, due to slower client-side execution. Obviously, this is the cost to lessen the privacy concern. We take a closer look at the performance loss of partial inference shortly.

We measured the breakdown of the inference time to observe the overhead of the snapshot. Fig. 7 shows the result. 'C' and 'S' in parentheses indicate the client and the server, respectively (i.e., *Snapshot Capture (C)* means the time to capture a snapshot at the client). In all configurations, the overhead of capturing and restoring a snapshot is negligible compared to the DNN execution at the client or at the server, meaning that the snapshot overhead is not serious when offloading computation-intensive DNN execution. The most dominant part of the inference time is the server execution time in both

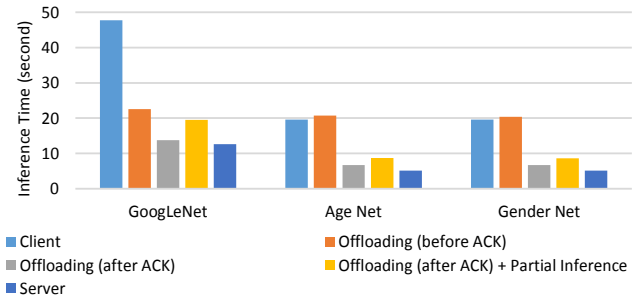

Figure 6: Execution time of inference in three web apps

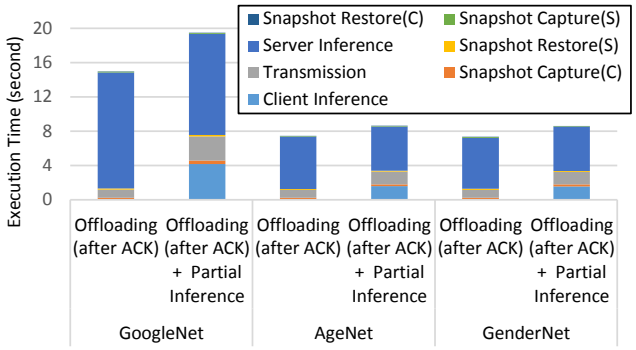

Figure 7: Breakdown of the inference time

configurations, because the server performs most of the DNN inference. The server execution time itself will be sharply reduced in the near future, since ML web frameworks are starting to use GPUs for DNN execution [27] [28] (e.g., webGL can give ~80x speedup for DNN inference [27]).

B. *Performance Impact of Partial Inference*

This section investigates the performance impact of partial inference. Fig. 8 shows the inference time of each DNN when applying partial inference with different offloading points. The X-axis indicates the last layer of the client-side partition. For example, *1st_conv* means that the client executes the DNN from the input layer to the first convolution layer and then offloads the remaining DNN execution to the server. The server performs the rest of the layers and sends the result back to the client. As expected, offloading with *partial* inference leads to lower performance than offloading of *full* inference (offloading with *Input* in Fig. 8) because we need to compute more at the client device equipped with poor hardware.

One interesting observation in Fig. 8 is that the inference time does not increase continually as more layers are executed at the client. In fact, the inference time decreases when the offloading point moves from a conv layer to a pool layer. One reason is the amount of computation required for each layer. For instance, the result of offloading at conv layer shows long client-side inference time because conv layer has high computation requirements to run convolution functions. On the other hand, pool layer requires a simpler computation (*max* function), so the client execution time increases slightly with it. Another reason is that the size of feature data varies for DNN layers, affecting the transmission overhead. When we examine the size of the feature data produced at each offloading point, we found that the size surges in the conv layers due to many convolution filters. On the contrary, the pool layers reduce the size of feature data because of the max pooling operation (e.g., in GoogleNet, the size of feature data is 14.7MB in *1st_conv* while it is 2.9MB in *1st_pool*; other models also show a similar size behavior). Since the feature data takes most of the total transmitted data, the data transmission time heavily depends on the feature data. Based on these observations, the first pool layer (*1st_pool*) appears to be the best offloading point that can minimize the

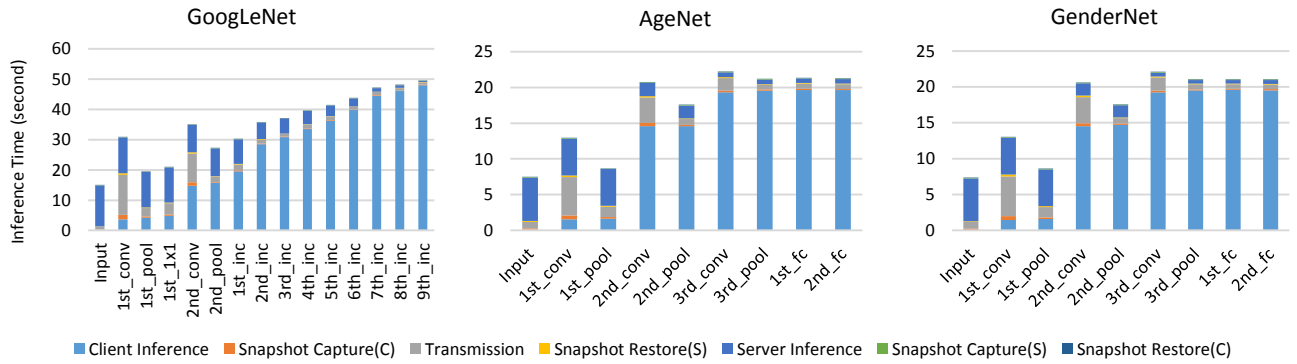

Figure 8: Inference time with partial inference at various offloading points

inference time, yet still denaturing the input data. In fact, the partial inference result in Fig. 6 was based on offloading at *1st_pool* layer.

C. *Overhead of VM-based Installation*

We implemented the VM-based installation described in Section 3.B based on the open source project for VM synthesis [26]. We create a VM overlay which contains the offloading programs, the libraries, the browser, and the DNN model for the app. The VM overlay is sent to the server and synthesized with a base VM image of Ubuntu 12.04. The server runs the synthesized VM instance using a virtual machine monitor (QEMU and KVM) [14]. We measure the size of the VM overlay and the time to perform VM synthesis (including the time to upload VM overlay and the time to synthesize a VM instance), which will be the installation overhead needed when our offloading system is not pre-installed at the edge server.

Table 1 shows the result. VM synthesis takes a substantial overhead (around 19 or 24 seconds), and it is mainly due to the transmission of the large VM overlay of 65 or 82 MB; the overlay, compressed with LZMA algorithm, consists of the browser (~45MB), the libraries (~54MB), the offloading server program (~1MB), and the model (rest) before compression. For comparison Table 1 also shows the overhead of snapshot-based offloading with and without the model pre-sending, which includes the time to save/transmit/restore the snapshot just before executing the offloaded event handler. Table 1 indicates that once the VM synthesis is done for dynamic installation, snapshot-based offloading can swiftly migrate the app state (less than a second), because the large DNN model is already uploaded to the server, while the snapshot itself is tiny.

Let us assume that our offloading system were installed on the edge server in advance. Table 1 indicates that even if pre-sending were not used, the overhead of the first snapshot-based offloading, which needs to deliver the model as well, will be reasonable (around 7 or 12 seconds), which is much smaller than the VM synthesis. The next offloading requests will require a tiny overhead, equivalent to the pre-sending case, since the model is already available at the edge server. These results imply that snapshot-based offloading would work relatively efficiently, whether the offloading system is pre-installed or dynamically installed on demand.

TABLE I. OVERHEAD OF VM-BASED INSTALLATION FOR SNAPSHOT-BASED OFFLOADING

Configuration		GoogleNet	Agenet	Gendernet
VM synthesis	Synthesis time (Second)	19.31	24.29	24.31
	VM overlay (MB)	65	82	82
Snapshot-based Offloading (w/ pre-sending)	Migration time (Second)	0.60	0.34	0.34
	Snapshot except feature data (MB)	0.09	0.02	0.02
Snapshot-based Offloading (w/o pre-sending)	Migration time (Second)	7.79	12.07	12.07
	Snapshot except feature data (MB)	27	44	44

V. RELATED WORK

Computation offloading has been a long standing issue in the field of mobile computing. *Cyber foraging* is to offload mobile client's computations to nearby servers (also known as surrogates) [21]. *Cloudlet* has been proposed as decentralized cloud infrastructure to realize the cyber foraging [4]. To use the cloudlet in a transient way, VM-based customization has been widely studied [4] [14]. Our snapshot-based offloading is different from the VM-based customization, since we achieve offloading by sending the app execution state instead of VM.

Gabriel is a recent work on offloading computations to edge servers using VM-based customization, to implement mobile cognitive assistance [8]. Gabriel customizes an edge server with its cognitive engines encapsulated within the VM image, making the server execute cognitive services, such as face recognition or augmented reality. Snapshot allows the edge server to execute any computation of any app, instead of specific cognitive services included in the VM image.

There are other offloading approaches which migrate app execution. MAUI offloads the app execution to a server where the app executable is pre-installed [22]. When a client transfers control to the server, MAUI captures the application state and transmits it to the server. The server restores the application state, resumes the execution using the pre-installed app executable, and sends the result back to the client. This workflow has been widely used in other offloading frameworks, such as CloneCloud [23] and

ThinkAir [24]. Tango is another offloading approach that runs replicas of application both on the client and the server and displays the faster one's result on the screen to reduce the user-perceived latency [29]. Our approach is different from them in the sense that we do not require the app executable to be pre-installed at the server, because the snapshot is an executable, self-contained web app by itself. This makes snapshot-based offloading more appeal to mobile users who want to offload to any nearby edge server.

Recently, there have been studies that install the same DNN models at the client as well as at the server, and execute the models partly by the client and partly by the server to trade-off accuracy and resource usage [25] or to improve performance and energy saving [16]. However, both require the DNN models to be pre-installed at the server, thus not appropriate for edge computing where the offloading requests can be made to any edge servers. Our work is basically an approach to send the DNN model to the edge server on-demand, thus requiring no pre-installation. Also, we partition DNN to improve privacy.

VI. Summary and Future Work

We proposed snapshot-based offloading as a new approach to running ML web apps on embedded devices in the edge computing environment. We found that snapshot-based offloading works for real DNN-based web apps, with a promising result of performance and privacy. Actually, our work is the first step to *app-level customization* for generic edge servers, which allows lighter customization than existing *VM-level customization*. Once customized with the first offloading, however, it is an issue how to simplify the snapshot creation/transmission/ restoration for future offloading using the data and code left at the server from the first offloading. This is left as a future work.

Acknowledgment

This work was supported by Basic Science Research Program through the National Research Foundation (NRF) of Korea funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B2005562).

References

[1] Satyanarayanan, M. (2017). The emergence of edge computing. Computer, 50(1), 30-39.

[2] Borcoci, E. (2016, August). Fog Computing, Mobile Edge Computing, Cloudlets which one?. In SoftNet Conference.

[3] Bonomi, F. et. al, (2012, August). Fog computing and its role in the internet of things. In Proceedings of the first edition of the MCC workshop on Mobile cloud computing (pp. 13-16). ACM.

[4] M. Satyanarayanan, P. Bahl, R. Caceres and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," in IEEE Pervasive Computing, vol. 8, no. 4, pp. 14-23, 2009.

[5] N. Chen et. al, "Dynamic Urban Surveillance Video Stream Processing Using Fog Computing," *2016 IEEE Second International Conference on Multimedia Big Data (BigMM)*, Taipei, 2016.

[6] Makinen, Olli. (2015). Streaming at the Edge: Local Service Concepts Utilizing Mobile Edge Computing. 1-6.

[7] Lai, Zeqi & Charlie Hu, Y & Cui, Yong & Sun, Linhui & Dai, Ningwei. (2017). Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices.

[8] Kiryong Ha et. al,. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services* (MobiSys '14). NY, USA, 68-81.

[9] JinSeok Oh et. al, 2015. Migration of Web Applications with Seamless Execution. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (VEE '15). ACM, New York, NY, USA, 173-185.

[10] Hyuk-Jin Jeong and Soo-Mook Moon, "Offloading of Web Application Computations: A Snapshot-Based Approach," *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, Porto, 2015, pp. 90-97.

[11] Jin-woo Kwon and Soo-Mook Moon. 2017. Web Application Migration with Closure Reconstruction. In Proc. the 26th International Conference on World Wide Web (WWW '17). Australia.

[12] WebKit Open Source Project, www.webkit.org

[13] Szegedy, C. et. al, (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[14] Kiryong Ha et. al. 2013. Just-in-time provisioning for cyber foraging. In Proceeding of the 11th annual international conference on Mobile systems, applications, and services *(MobiSys '13).* ACM, New York, NY, USA, 153-166.

[15] Levi, G., & Hassner, T. (2015). Age and gender classification using convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops.

[16] Yiping Kang et. al,. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. SIGARCH Comput. Archit. News 45, 1 (April 2017), 615-629.

[17] Mahendran, A. & Vedaldi, A. (2014) Understanding Deep Image Representations by Inverting Them, http://arxiv.org/abs/1412.0035

[18] Hemminger, S. (2005, April). Network emulation with NetEm. In Linux conf au (pp. 18-23).

[19] Caffe.js framework. https://chaosmail.github.io/caffejs

[20] Jia, Y. et. al, (2014, November). Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia.

[21] M. Satyanarayanan. Pervasive computing: vision and challenges. Personal Communications, IEEE, 8(4):10–17, 2001.

[22] Cuervo, E. et. al, (2010, June). MAUI: making smartphones last longer with code offload. In Proceedings of the 8th international conference on Mobile systems, applications, and services. ACM.

[23] B.-G. Chun et. al, "CloneCloud: elastic execution between mobile device and cloud," in Proc. ACM European Professional Society on Computer Systems (EuroSys'11), Salzburg, Austria, Apr. 2011.

[24] Kosta, S. et. al, (2012, March). Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In Infocom, 2012 Proceedings IEEE (pp. 945-953).

[25] Han, S. et. al, (2016, June). Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services. ACM.

[26] Elijah cloudlet system. https://github.com/cmusatyalab/elijah-cloudlet

[27] keras.js website. https://github.com/transcranial/keras-js

[28] deeplearn.js website. https://deeplearnjs.org/

[29] Gordon, Mark S., *et al*. "Accelerating mobile applications through flip-flop replication," in Proc. the 13th International Conference on Mobile Systems, Applications, and Services. ACM, 2015.