2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) | 978-1-7281-7002-2/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICDCS47774.2020.00114

PerDNN: Offloading Deep Neural Network Computations to Pervasive Edge Servers

Hyuk-Jin Jeong[*], Hyeon-Jae Lee, Kwang Yong Shin, Yong Hwan Yoo, Soo-Mook Moon[*]
Department of Electrical and Computer Engineering
Seoul National University
Seoul, South Korea
jinevening@snu.ac.kr, thlhjq@snu.ac.kr, kwangshin@altair.snu.ac.kr, yyh729@snu.ac.kr, smoon@snu.ac.kr

***Abstract*—Emerging mobile applications, such as cognitive assistance based on deep neural network (DNN), require low latency as well as high computation power. To meet these requirements, *edge computing* (also called *fog computing*) has been proposed, which offloads computations to edge servers located near mobile clients. This paradigm shift from cloud to edge requires new computing infrastructure where edge servers are pervasively distributed over a region.**

This paper presents PerDNN, a system that executes DNNs of mobile clients collaboratively with pervasive edge servers. PerDNN dynamically partitions DNN computation between a client and an edge server to minimize execution latency. It predicts the next edge server the client will visit, calculates a speculative partitioning plan, and transfers the server-side DNN layers to the predicted server in advance, which reduces the initialization overhead needed to start offloading, thus avoiding cold starts. We do not incur excessive network traffic between edge servers, though, by migrating only a tiny fraction of the server-side DNN layers with negligible performance loss. We also use GPU statistics of edge servers for DNN partitioning to deal with the resource contention caused by multi-client offloading. In the simulation with human trace datasets and execution profile of real hardware, PerDNN reduced the occurrence of cold starts by up to 90%, achieving 58% higher throughput when clients change their offloading servers, compared to a baseline without proactive DNN transmission.

I. Introduction

Future mobile applications, such as augmented reality [1], cognitive assistance [2], and cloud gaming [3], require not only intense computations but stringent latency constraints. Since cloud servers in distant data centers have difficulty in meeting the latency requirement, new computing infrastructure called *edge servers* (also called cloudlets [4] or fog nodes [5]) has been proposed, which offer high bandwidth, low-latency access to storage and computing resources at the edges of the network, e.g., Wi-Fi APs [4] or small cells [6]. IEEE recently adopted OpenFog Reference Architecture (OpenFog RA) as an official standard [7], which describes a generic architecture that distributes computing servers closer to data sources such as IoT devices.

OpenFog RA presents the sketch of a large-scale edge network composed of multiple inter-connected servers installed in smart cities [7]. One possible form of such a network is *public edge servers*, a cluster (or hierarchy) of edge servers that perform computations of user applications in a public place, e.g., airport or subway. A key attribute of public edge servers is *versatility*, which can perform general computations of any authorized clients, enabling pervasive computing environments for users running various applications. This is in sharp contrast to *private edge servers*, which provide a set of pre-installed services only to a few clients, resulting in limited use of edge servers.

A promising application of public edge servers is to offload computations for *deep neural network* (*DNN*), which has a great model diversity as well as a high computation requirement. For example, *mobile cognitive assistance* (*MCA*) [2] helps blind people by recognizing objects seen by wearable glasses and telling the objects to the owner. Recent techniques for object recognition use compute-intensive DNNs [8] [9], so such a service can offload the DNN computations to a nearby edge server to improve app performance and energy consumption of wearable glasses. An important aspect of DNNs is that they can be modified (or replaced) after deployed. If MCA needs to recognize a new object (e.g., a new family member), it may require new model structure with the additional output label and the re-training with the new dataset. Also, the model can be continuously trained using locally collected data to specialize in the target environment [10]. These situations require each client to have its own model trained by the client's specific task and data, which heightens the need for versatile edge servers where clients can dynamically deploy their custom DNN models and offload execution. The purpose of this paper is to explore the feasibility of such a public edge computing system for running users' custom DNN models.

There are two primary challenges to achieve high and stable performance when offloading DNN computations to public edge servers. First, we need to deal with the performance drop of DNN offloading at *cold start* for a new edge server. To offload DNN execution to a new edge server, the DNN model has to be first deployed to the server, which significantly delays the time to start offloading. Although runtime provisioning of edge servers has been extensively studied in edge computing communities [11] [12] [4], they mainly focus on optimizing the migration of VM or container, not DNN models. A recent work named IONN [13] mitigates the performance drop by incrementally transmitting DNN layers and by partially offloading the execution of the transmitted layers, yet still

[*]Corresponding authors.

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

suffers from low performance during the initial stage of model transmission (see Section 2.B). Secondly, whenever a client offloads its DNN computations, it should offload to the best edge server among the nearby ones, considering many *dynamic factors*, such as workloads of edge servers or network status, to minimize DNN execution latency. This is especially true when multiple clients simultaneously offload DNN computations to a single server, so the server might suffer from GPU contention, which affects the DNN execution performance.

To tackle those challenges, we propose *PerDNN*, a system that manages the offloading of DNN execution between mobile users and many inter-connected edge servers. PerDNN selects the best edge server to offload DNN execution using a partitioning algorithm based on runtime states, such as GPU statistics of edge servers and network conditions, as well as hyperparameters of DNN models. After edge server selection, PerDNN dynamically deploys the user's DNN layers to the edge server and executes the DNN model collaboratively (partially at the client and partially at the server), to minimize the DNN execution time. To avoid the cold start that occurs when a user moves to a different edge server, PerDNN periodically predicts the next edge server to visit based on the user's recent trajectory, calculates a speculative partitioning plan between the client and the predicted server, and proactively migrates the server-side DNN layers of the plan to the next edge server. This allows the user to immediately start offloading DNN execution when visiting the predicted edge server. To reduce the network traffic, we select and migrate only a fraction among the server-side layers for the crowded edge servers with heavy network traffic, which sharply cuts the network traffic with negligible performance degradation. To our knowledge, PerDNN is the first study to 1) exploit GPU information for DNN partitioning and 2) perform real-time proactive caching in the context of edge computing.

To evaluate PerDNN, we simulated edge computing scenarios where more than a hundred users offload DNN computations to edge servers dispersed in a smart city while they are on the move. For simulation, we used two open source mobility datasets collected from Beijing [14] and KAIST [15], and execution profiles of real embedded boards and desktop servers. Using a linear SVR model, which showed high accuracy among various trajectory prediction algorithms, PerDNN predicted the next move of the clients and proactively transmitted their DNN layers to the edge servers around the predicted location. It reduced the occurrence of cold starts by 70$\sim$90% and achieved 58$\sim$97% higher DNN query throughput when mobile users change their offloading servers, compared to a baseline with no proactive transmission. Also, we could reduce 43$\sim$67% of the peak backhaul traffics needed for proactive migration by migrating a fraction of a DNN model for crowded servers, with 1$\sim$2% of performance loss.

The rest of this paper is organized as follows. Section 2 introduces the background and the motivation of this paper. Section 3 describes the design and implementation of PerDNN. We evaluate our system in Section 4, survey related works in Section 5, and conclude in Section 6.

II. Background and Motivation

In this section, we give a brief explanation on DNN and introduce a cold start issue raised when offloading DNN execution to a public edge server.

A. Deep Neural Network (DNN)

DNN is organized with DNN layers. Each DNN layer receives input tensors from the previous layers, performs its operation, and delivers the output tensor to the next layers. A DNN layer contains two types of parameters, one is *hyperparameters*, whose values are fixed during training a DNN model, and the other is *weights*, whose values are updated by learning algorithms (e.g., gradient descent) with training data. The training of a DNN model requires a massive amount of memory and computational resources, so it is usually performed in cloud datacenters. The trained DNN model is deployed to clients and used to infer outputs from new input data. In this paper, we only deal with the offloading of *DNN inference*, which has a direct impact on user experience in mobile applications.

B. Overhead of Deploying DNN Models

To offload DNN execution from a client to an edge server, the client's DNN model has to be present in the server. Commercial edge computing solutions of major cloud providers (*Amazon GreenGrass, Microsoft Azure IoT Edge, Google Cloud IoT Edge*) let users save their DNN models in the cloud and make an edge server download the model before executing it. However, since modern DNN models for complex tasks typically consist of numerous weights, users likely will wait for a quite long time to download full DNN models. For example, *Inception 21k* [9], a popular DNN model for image classification with size of 128 MB, is completely downloaded in $\sim$18 seconds under the global average download speed ($\sim$57.91 Mbps [16]), which is hardly tolerable for today's mobile users. The high overhead of model deployment makes it difficult to offload DNN execution to public edge servers on demand, especially when the user moves frequently.

A recent study named IONN [13] mitigates the waiting time for model deployment by *partitioning* a DNN model between a client and a server and by *incrementally uploading* the server-side layers from the client to the server (IONN assumed that a model is saved in the client and uploaded to the edge server for offloading [13]). IONN partially offloads the execution of uploaded layers before the full model is transmitted, so DNN execution performance is gradually improved as more layers are uploaded to the server. This deployment strategy improves DNN execution performance during uploading of DNN layers, but initial performance remains low when a client just starts uploading DNN layers, as the client has to execute most of DNN layers by itself.

Fig. 1 demonstrates the performance drop of IONN, which happens when a client starts to offload DNN computations to a new edge server. We measured the execution times of 40 consecutively generated DNN queries on *Inception21k* [9]. Each query was raised 0.5 second after the previous query is

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

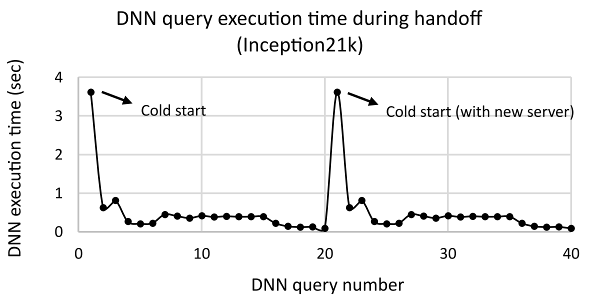


Fig. 1. DNN execution time while a user moves from one edge server to another.

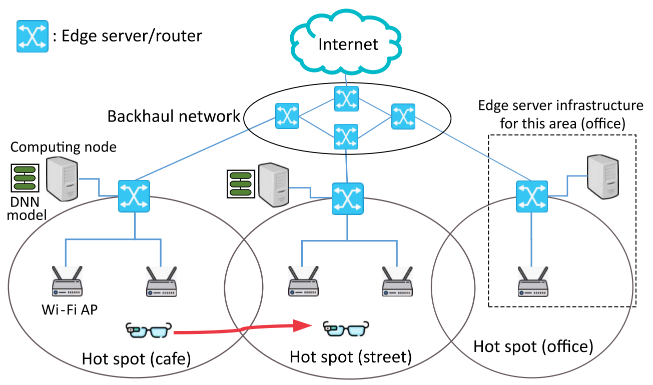


Fig. 2. Edge server environment based on Wi-Fi APs.

executed. The client device was an embedded board (ODROID XU-4 [17]), and the server was a desktop PC equipped with a high-end GPU (Titan Xp). For the first query, the client executed all DNN layers at the local device as no layers had been uploaded to the edge server yet, so the execution time was quite high. DNN execution time decreased as more layers were uploaded, but soared at the 21st query, where the client changed its offloading server. Although DNN execution time decreased again rapidly due to incremental offloading, the spike of execution time at the start of offloading would harm users' mobile experience, e.g., frame drops in video analytics whenever a user moves from one edge server to another. Mobile users who frequently change their target edge servers would be especially vulnerable to the fluctuation.

III. Proposed Offloading System: PerDNN

In this section, we present the design and implementation of PerDNN, which reduces cold starts by predicting the movement of mobile users and proactively migrating DNN layers to the next edge server to visit. We first describe our edge server environment and the overall architecture of PerDNN. Next, we delve into how we predict the next edge server and partition a DNN model.

A. Edge Server Environment

Fig. 2 illustrates our edge server environment. We considered a general wireless local area network (WLAN) where mobile users connect to nearby Wi-Fi APs in public places, e.g., cafe, street, or office. Next, we envisioned edge server infrastructure integrated with WLAN, where computing nodes, equipped with accelerators such as GPUs, are located near the hotspots. The edge server of each hotspot routes data between the users and the nearby computing node, so hotspot users can access the computing node with low latency. Edge servers are inter-connected through *backhaul network*, which has been used for file caching [18] [19] or service migration [11]. In this paper, the backhaul network is used to transmit DNN layers between edge servers.

A hot spot client (smart glasses in Fig. 2) can offload DNN execution to the computing node after deploying its DNN model (or layers) to the node. We assumed the model is uploaded from the client to the node (like IONN [13]), but it is also possible to make the node download the model from the cloud. When the client moves to another hotspot, it can either (1) offload DNN execution to the new computing node in the current hotspot, as previously or (2) keep connection with the previous server and route input/output data through backhaul networks. In this paper, we focus on the first case, because routing leads to the sub-optimal offloading with increased latency and constantly consumes backhaul traffics while the client offloads computations. It should be noted that despite these drawbacks, routing can be useful in some cases (e.g., load balancing [20]), but we leave the possibilities as the future work.

B. Overall Architecture

Fig. 3 shows the overall architecture of PerDNN. The proposed system consists of a set of inter-connected edge servers and a *master server*, which controls the deployment and the collaborative execution of DNN models. The system works as follows.

When a mobile client first connects to one of the edge servers, it uploads two types of information to the master server: *DNN profile* and *current trajectory*. *DNN profile* includes the types and hyperparameters of DNN layers, which are used to partition a DNN model. *DNN profile* does not contain the weights of layers (the heaviest part of a DNN model), so it can be quickly uploaded to the master server. *Current trajectory* is the recent locations of the client during a certain time period, gathered from GPS or Wi-Fi positioning system. The client periodically transmits its current trajectory to the master server, so the master server can keep track of the latest trajectory of the client.

Using the *DNN profile* and the *current trajectory* of the client, the master server controls the following actions: 1) collaborative DNN execution and 2) proactive migration of DNN layers to the next visited edge server.

1) Collaborative DNN execution: DNN partitioner in the master server creates a partitioning plan for a client, which specifies the execution location of each DNN layer and how to upload DNN layers. DNN partitioning is further explained in Section 3.C. The partitioning plan is delivered to the client, and the client incrementally uploads DNN layers to the edge

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

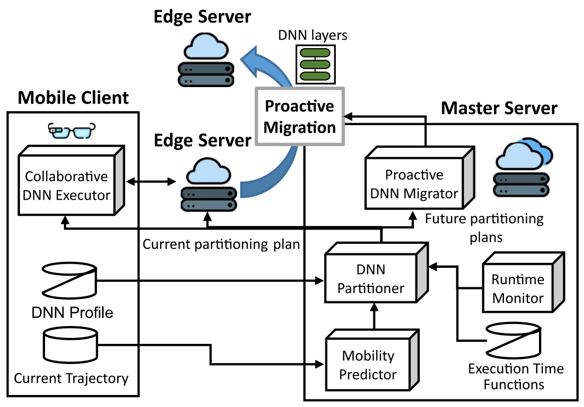


Fig. 3. Overall architecture of proposed system.

server according to the plan. When a DNN query is raised, the client executes layers one by one until the execution reaches the uploaded layer, and sends the input of the uploaded layer to the edge server. The edge server executes the uploaded layers and returns the result to the client. The client continues to execute the rest of the layers, if any. Since the partitioning plan is used for this current DNN execution, we call the plan *current partitioning plan*.

2) Proactive DNN migration: While the client and the edge server collaboratively execute a DNN model, the master server predicts the next location of the client using the client's current trajectory. Then, it finds edge servers around the predicted location by finding nearby hotspots in the Wi-Fi database such as WiGLE [21] under the assumption that the master server has the mapping between edge servers and Wi-Fi hotspots. The master server then creates partitioning plans for the edge servers near the predicted location; these plans are derived for future execution, so we call them *future partitioning plans*. The server-side layers of the future partitioning plans are transmitted from the current edge server to the corresponding edge servers. If the current edge server does not have all of the server-side layers, it sends layers as many as possible, so the client can upload only the rest of the layers when it visits the predicted server. After receiving DNN layers, edge servers keep the layers for a certain duration (*TTL*: *Time To Live*) and discard them after *TTL*. *TTL* is reset when another server attempts to send the DNN layers of the same client to the server, so the layers already existing at the server are not sent again, thus avoiding duplicate transmissions.

PerDNN periodically repeats the above processes to update partitioning plans according to runtime states and to keep sending the DNN layers to the edge servers near the mobile clients. When a client visits an edge server, PerDNN creates the *current partitioning plan*, as mentioned in Section 3.B.1; it should be noted that the plan is based on the current runtime states (server workloads and network conditions), so it might be different from the future partitioning plan made previously for proactive migration to this server. If the server already received all (or parts) of the server-side layers from other servers, the client can immediately start offloading the execution of those layers. If there are some missing server-side layers, the client will upload the remaining layers to complete the current partitioning plan. If the server does not have any server-side layer of the current partitioning plan, the client incrementally uploads the server-side layers from scratch. Interestingly, we found that migrating only a tiny fraction of the server-side layers can improve the performance substantially, which we can exploit to reduce the network traffic (see Section 4.A).

C. DNN Partitioning

1) GPU-aware Execution Time Estimation: The objective of DNN partitioning is to create a partitioning plan, which determines the execution location of DNN layers to minimize execution latency. Previous studies on DNN partitioning [13] [22] have already introduced algorithms to find the best partitioning plan with minimum execution time based upon the estimated execution time of each DNN layer and the time to transmit tensors. So far, however, no previous study has considered the congestion of GPU, which is critical to DNN execution performance, when partitioning a DNN model, thus having difficulty when multiple clients simultaneously offload DNN execution to an edge server equipped with a shared GPU. PerDNN uses GPU information when estimating layer execution time, so it can derive a more accurate partitioning plan than previous approaches in case of multi-client offloading.

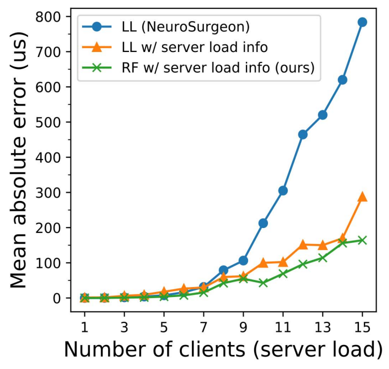
However, building an estimation model for layer execution time is extremely challenging, because it is affected by numerous factors. When a server concurrently executes DNN inferences of multiple clients, the execution time is influenced by interference in shared GPU or system resources such as streaming multiprocessors, GPU memory, and PCIe bus. Various GPU sharing schemes (temporal sharing [23], spatial sharing [24], or a hybrid of them) make it more difficult to predict the execution time. Estimating layer execution time considering all these factors is extremely complicated, and not feasible in general. Therefore, we take a practical approach that does not require any prior knowledge of hardware details or GPU scheduling policies but use GPU information to estimate resource contention.

PerDNN predicts layer execution time based on GPU statistics as well as layer hyperparameters. We assumed edge servers can measure GPU statistics[1] including kernel/memory utilization, GPU memory usage, and GPU temperature without significant overhead. So, before deriving a partitioning plan, the master server pings to an edge server to obtain the current server workload and then estimates layer execution time using the *execution time estimator* of the server based on the current server workload.

The *execution time estimator* of each edge server is trained offline using the dataset generated by profiling the execution

[1]Kernel/memory utilization and GPU temperature were measured with nvml, Nvidia management library. Kernel/memory utilization means the percentage of time spent for kernel execution or memory operation over the past sample period (between 1 second and 1/6 second)

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.



Feature	Importance (%)
# of clients	35.3
GPU Kernel util	24.2
Memory op util	11.5
GPU temperature	6.9
input z	5.2
kernel size	5
input y	3.8
input x	3.6
GPU memory usage	2.4
filter size	2.0
stride	0.1

Fig. 4. Mean absolute errors of execution time estimation with different server workloads (conv layer).

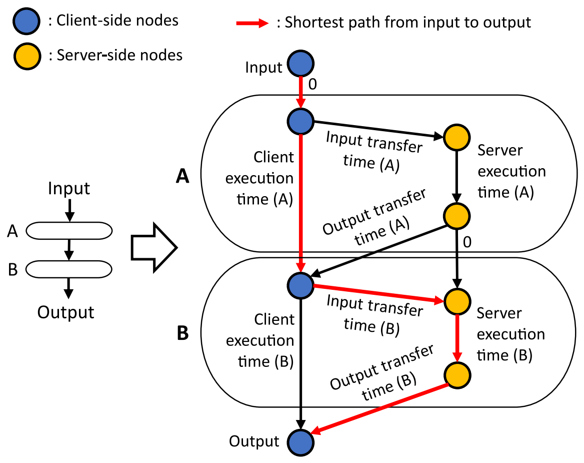


Fig. 5. Graph-based DNN Partitioning Algorithm.

time of a DNN layer while concurrently running DNN inferences issued by multiple clients. During the profiling, server workload is changed by adjusting the number of clients; we extended *perf_client* in NVIDIA TensorRT inference server [25] to control the concurrency level and to measure the execution time. At the same time, the server records its GPU statistics whenever receiving a DNN request from a client. Using the dataset, the edge server trains random forest models for each layer type (conv, fc, etc.) that predict the execution time given the server workload and the layer hyperparameters.

Fig. 4 shows the mean absolute error (MAE) of the estimated execution time for a convolution layer. For comparison, we plotted the MAE of the estimation model of the latest approach (*NeuroSurgeon* [26]), which only uses layer hyperparameters to train linear/logarithmic regression models (represented as *LL*); different models were trained for each server load (≈ number of clients), as described in their paper. The result shows that MAE of *LL* surges as the number of clients increases. The MAE of a single layer is insignificant (at most ∼800 us), but modern DNN models often use hundreds of layers, so the aggregate error of the entire model would be substantial.

To test if the GPU information improves the estimation, we trained the same *LL* models but with GPU statistics as well as layer hyperparameters (represented as *LL w/ server load info*). *LL w/ server load info* showed much lower MAE than *LL* when many clients send requests, which implies that GPU information is useful for estimating layer execution time under heavy workloads. Our method (*RF w/ server load info*) showed even less MAE than *LL w/ server load info*, suggesting that a random forest is better than linear/logarithmic models to learn the non-linear relationship between execution time and workload features (# of clients, kernel/memory utilization, GPU temperature). The right side of Fig. 4 shows the *importance* of each feature in our random forests [27]. It indicates that the workload features were more important than layer hyperparameters when estimating layer execution time.

2) Partitioning Algorithm: Given the estimated layer execution time, the optimal partitioning plan leading to the minimal execution time can be found using a graph-based partitioning algorithm [13]. Figure 5 illustrates the algorithm with an example DNN composed of two layers. The DNN is transformed to a directed graph which expresses the data flow between a client and a server during DNN execution. The blue and yellow nodes indicate client-side and server-side nodes, respectively. Edge weights, which represent the time taken for layer execution and data transfer, are determined as follows. Server-side layer execution time is gained from the execution estimator of the edge server, as explained in Section 3.C.1. Client-side layer execution time is gained from the DNN profile. Input/output transfer time of each layer is calculated by dividing the input/output size saved in the DNN profile by the runtime network speed. Since the sum of edge weights on the path from the input to the output is the time to execute the DNN model, the fastest execution path is the shortest path from the input to the output. In Figure 5, the shortest path (red arrows) indicates that running layer A at the client and layer B at the server is the fastest way to execute the DNN.

By applying the algorithm to all edge servers visible to the client, the master server can find the best edge server with the minimum execution time and the corresponding execution locations of the DNN layers, which is the *current partitioning plan*. Since the layer execution time of each server is estimated considering the GPU statistics, the workload is automatically balanced; crowded servers with long execution time would not be selected for offloading. To derive *future partitioning plans* for proactive migration, the master server applies the same partitioning algorithm to the edge servers within a certain distance (50 m or 100 m in our evaluation) from the predicted location. We use the current GPU workloads to estimate the execution time for *future partitioning plans*, under the assumption that the GPU workloads of edge servers do not change so abruptly in a close future (less than 30s, as determined in Section 3.D). The server-side layers are proactively migrated to all edge servers within the distance, so that the client can immediately start offloading when visiting

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

one of the servers.

After the execution location of each layer is determined, we need to decide the order of uploading the server-side layers, i.e., which layers should be uploaded to the server first. Our strategy is to send layers with high offloading benefits first [28]. We defined the *efficiency* of layers as the latency reduced by offloading the layers divided by the size of the layers. We create the *partitions* of the server-side layers, which are all possible successive layers in the server-side layers, and calculate the *efficiency* of each partition. Then, we decide to upload a partition with the highest *efficiency* first and update the *efficiency* of the remaining partitions. The same process is repeated until all server-side layers are uploaded. We use the same algorithm for proactive migration as well, to determine the order of sending layers from the current edge server to the next edge servers. This *efficiency*-based algorithm makes the next edge servers receive high-efficiency layers first, so that the client can offload the execution of those layers even though the whole server-side layers were not transmitted yet.

D. Mobility Prediction

The purpose of mobility prediction is to predict the location of a mobile user to determine the next visited edge servers. Since PerDNN continuously makes predictions in real time, its prediction mechanism has to be lightweight. Also, the prediction should capture the point where the client enters the service area of the next server, so it needs to be conducted at short intervals. To meet these requirements, PerDNN predicts the user's next location based on the user's *recent trajectory*, which can be easily collected in modern mobile devices and highly correlated with the user's next movement [29]. We assumed a client periodically collects its location (x, y coordinates) every time interval t and sends n most recent locations to the master server. The master server predicts the next location of the client after the interval t.

To achieve high prediction accuracy and system efficiency, the values of n and t have to be carefully determined. Trajectory length (n) directly affects the prediction accuracy, e.g., if n is too small, the prediction will be inaccurate due to the lack of previous location information. Time interval (t) affects the number of *futile predictions* as well as prediction accuracy; *futile predictions* are predictions made while the client stays in the same edge server, which do not contribute to any performance improvement but waste the resource to make predictions, e.g., mobile device energy and backhaul traffics. If we make predictions too frequently (i.e., t is small), the master server will make many futile predictions before a client leaves the server. If we increase t to reduce futile predictions, however, the prediction error will increase, because the master server will predict the client's location of a too distant future.

To determine the values of n and t, we investigated their impacts on mobility prediction using the international-scale open source dataset named *Geolife* [14]. *Geolife* contains trajectories of mobile users with short measurement intervals (1∼5 seconds), so we could make datasets with different time intervals by sampling the trajectory data in a different

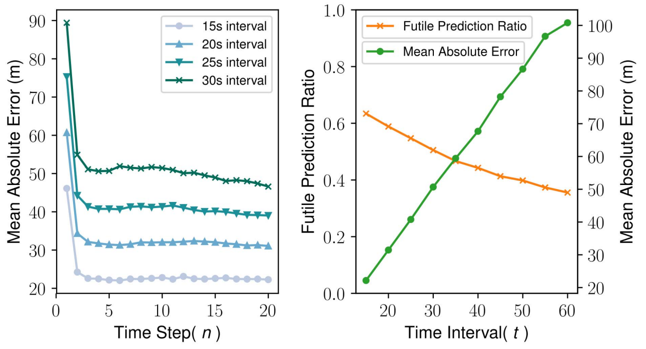


Fig. 6. Left: Prediction errors with different time steps. Right: Effects of time intervals on futile predictions and prediction errors.

rate. We trained *singular vector regression* (SVR) models that predict the mobility of the users in the datasets and tested the performance of the predictors.

The left side of Fig. 6 shows prediction errors (MAE) of the mobility predictors while changing the length of trajectories (n) used for prediction. For all time intervals (15, 20, 25, 30 seconds), the prediction error dropped when n is two, which implies that the locations of the recent two time steps are crucial to predict the location of the next time step; this finding is consistent with that of Song et al. [30]. The prediction errors slowly decreased (30s interval) or remained the same (the others) after five steps, so we set n as five in this paper.

The right side of Fig. 6 shows the effects of time interval t (unit: second) on prediction errors (MAE) and the ratio of futile predictions over all predictions. For the experiment, we divided the region of the dataset into a hexagonal grid and assumed an edge server is allocated in each cell with radius of 50 m (see Section 4.B.1 for more details about the environment). The result shows that the larger t, i.e., making predictions infrequently, reduces the futile prediction ratio but at the same time increases the prediction error, as expected.

Prediction accuracy and the number of futile predictions are highly dependent on the characteristics of mobile users, e.g., how fast a user moves or how long a user stays in one place. Therefore, we determine the value of t based on the *benefit* and the *cost* of proactive migration for each dataset, which have the following characteristics.

$$benefit \propto a \times (p - f) \tag{1}$$

$$cost \propto p \tag{2}$$

where p is the number of total predictions, f is the number of futile predictions, and a is the prediction accuracy when the predicted location is inside the service range of the next edge server. We calculated the benefit-to-cost ratios for different time intervals ranging from 15 seconds to 60 seconds, and selected the best t with the maximum benefit-to-cost ratio. The best t was 20 seconds in our settings with Geolife dataset.

After t and n are determined, the next step is to build a mobility prediction model, which predicts the next value

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

(x, y coordinates) in the trajectory given historical trajectory information. There exist a ton of previous studies on trajectory-based mobility prediction [30] [31] [29] [32]. We engineered three previous prediction algorithms (Markov, SVR, RNN) to suit our edge server environment, and used the one with high prediction accuracy and fast performance (linear SVR). Comparison of each algorithm is reported in section 4.B.2. Detailed implementation of each algorithm is explained below.

Markov: We implemented a Markov model based on several previous studies on mobility prediction [31] [32] [30]. The client's location (x, y coordinates) was discretized by mapping to the identifier of the closest edge server; edge servers are assumed to be distributed in a hexagonal grid (see Section 4.B.1 for more details). We created a variable-order Markov model, implemented as a prediction suffix tree [33], from the history of trajectories, based on the frequency of a sequence. Given a new user's recent trajectory, we search the longest matching pattern in the suffix tree. We multiply a $(0 < a \leq 1)$ to the length of the longest matching pattern, and use the sampled subsequence with that length to get prediction [34]. The subsequence ratio (a) was set to 0.7, which achieved the best accuracy in our datasets.

Support Vector Regression (SVR): We made an SVR model [35] which takes the array of x, y coordinates, the recent trajectory of a client, as an input and outputs the x, y coordinates of the client's next location. The x, y coordinates were normalized to standard scores before fed into the SVR model for training and testing. We compared SVR models with different kernel functions (linear, polynomial, rbf) using scikit-learn v0.20.3 and chose the best one (linear SVR) with the highest accuracy. The model parameters (epsilon, tolerance) were empirically determined.

Recurrent Neural Network (RNN): We made an RNN model using a long short-term memory (LSTM) cell [36], which has been widely used for mobility prediction [37] [38] [29]. The client's trajectory was transformed to a sequence of x, y coordinates normalized to standard scores. An LSTM cell reads an input sequence and produces a *latent vector* with size of 16$\sim$32 (depending on a dataset). The latent vector is delivered to an fc layer with no activation function which outputs the x, y coordinates of the predicted location. We used MAE as a loss function and the Adam optimizer [39] with learning rate of 0.001. Hyperparameters (number of LSTM cells, latent vector size, learning rate, and optimizers) were empirically determined by using grid search.

IV. Evaluation

In this section, we evaluate PerDNN in two setups. First, we examined the performance improvement that a single client can gain with proactive migration. Next, we conducted large-scale simulation where a number of mobile users offload DNN computations to public edge servers in the smart city.

The test application was *mobile cognitive assistance*, which continuously performs DNN inference to recognize objects for a visually-impaired person [2]. We assumed that a mobile client constantly generates a DNN inference query 0.5 seconds after the previous query is executed. We used three widely used DNN models for experiment. *MobileNet* [40] is a tiny DNN model designed to run on resource-constrained devices. *Inception* [9] and *ResNet* [8] are larger DNN models for a more complex task and higher accuracy. Table I shows the details of each model.

TABLE I
DNN models used for evaluation.

Name	# of Layers	Size (MB)	Description
MobileNet	110	16	MobileNet v1. Image classification among 1k classes [40]
Inception	312	128	Inception. Image classification among 21k classes [9]
ResNet	245	98	ResNet-50. Image classification among 1k classes [8]

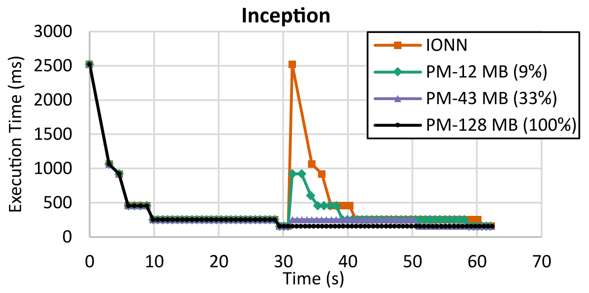
Our client board was an ODROID XU4 [17], equipped with ARM big.LITTLE CPU (2.0/1.5 GHz 4 cores) with 2 GB memory. We used an x86 desktop PC equipped with a quad-core CPU (i7-7700), Titan Xp GPU, and 32 GB memory as an edge server. The network speed between the client and the server was set to 50 Mbps for download and 35 Mbps for upload, the average speed of our lab Wi-Fi. Mobility predictor was implemented with a machine learning library named *scikit-learn* [27], and the rest of the system (DNN executor and DNN partitioner) was implemented based on a popular DNN framework named *caffe* [41].

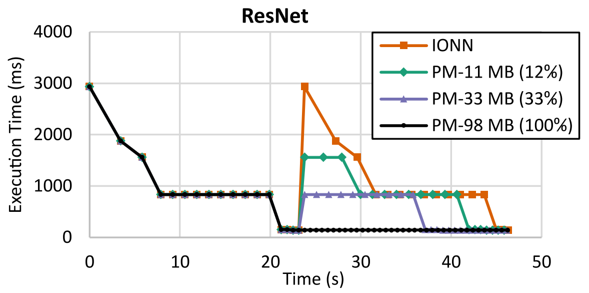
A. Performance Gain of Single Client

To evaluate the performance improvement from proactive migration, we measured the time to execute each DNN query raised by a client moving from one edge server to another. The baseline is the case where none of the layers were proactively migrated, so the client incrementally uploads the whole layers to both edge servers (denoted by *IONN*). *PM* denotes the proposed system, which proactively migrates layers from the previous server to the next server. We measured for the cases when all layers were migrated to the next server (100%), and when only partial layers were migrated (less than 100%). For the partial migration case, the non-migrated layers were incrementally uploaded when the client visits the next server. We also measured the amount of migrated layers in MB.

Fig. 7 shows the result. For all DNN models, the DNN query execution time of *IONN* spikes when a client changes its target edge server ($\sim$30 sec in *Inception*, $\sim$23 sec in *ResNet*, and $\sim$6 sec in *MobileNet*). The query execution time of *PM* increases relatively little, because *PM* can immediately offload the execution of proactively migrated layers when a client connects to the next edge server. Especially, *Inception* showed the remarkably lower peak execution time (2.8x speedup) when only 9% of the total model (12 MB) was sent to the server in advance. The dramatic speedup of *Inception* is due to its model structure, where compute-intensive convolution layers (having high *efficiency* for our efficiency-based uploading

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.





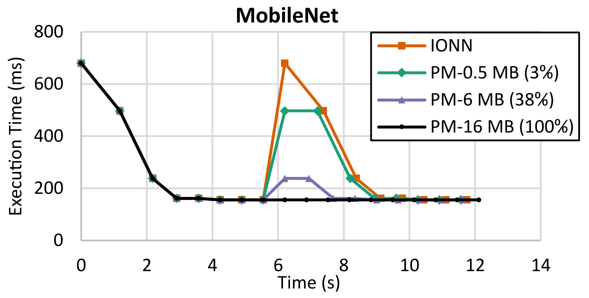


Fig. 7. DNN query execution time while a client changes its target edge server. IONN is the baseline. PM denotes the proposed system. The data size next to PM indicates the amount of DNN proactively migrated.

order mentioned in Section 3.C) are concentrated in the front part, so the execution performance was quickly improved by sending those layers first; other models have relatively lower-efficiency, evenly-distributed layers. These results indicate that migrating only a small fraction of a model can lead to significant performance improvement, which we can exploit to reduce the network traffic (see Section 4.B.4).

Next, we observed throughput gains achievable by proactive migration. We measured the number of executed queries while the client uploads all server-side layers to the server, i.e., throughput during uploading a DNN model. A *hit* case indicates that the server received all layers from another server in advance, so the client immediately offloads DNN execution; this represents the best performance of PerDNN. A *miss* case indicates that the edge server did not receive any DNN layer, so the client incrementally uploads its DNN layers from scratch, which represents the baseline (IONN [13]). Table II shows the throughput of each case. The throughput increase was small (4→5) in *MobileNet* due to its short uploading time, but for large DNN models, proactive migration significantly

TABLE II
NUMBER OF DNN QUERIES EXECUTED DURING UPLOADING OF A DNN MODEL (THROUGHPUT).

Data	MobileNet	Inception	ResNet
Uploading time (sec)	3.7	29.3	22.4
Executed queries of *miss* (IONN [13])	4	33	14
Executed queries of *hit* (ours)	5	44	34

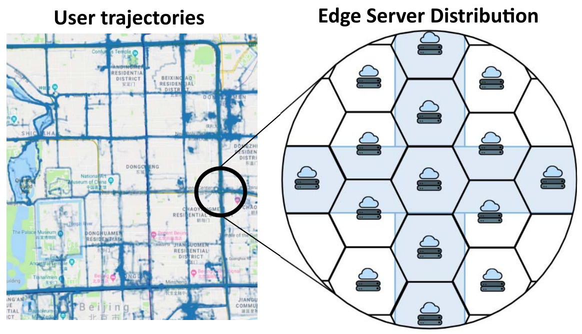


Fig. 8. User trajectories and edge server distribution. The blue part indicates where the user trajectories have passed.

improved the throughput (*Inception* (33→44) and *ResNet* (14→34)).

B. Large-Scale Simulation

1) Simulation Environment: We envisioned a public edge server environment where edge servers are pervasively distributed, so mobile users can access an edge server in any place. We set up such an environment based on two real-world mobility datasets listed below.

KAIST: KAIST is daily GPS tracks of students, collected every 30 seconds on campus [15]. To remove a few outliers who moved very far, we only used the data points in the rectangular area (1.5 km x 2 km) including the campus site.

Geolife: Geolife is a GPS trajectory dataset of 182 users, collected every 1∼5 seconds in multiple countries [14]. We only used the data inside the rectangular area (7.2 km x 5.6 km) encompassing the circular railway of subway line 2 in Beijing, China; the ranges of latitudes and longitudes are 39.900341∼39.950932 and 116.353370∼116.437765.

Fig. 8 visualizes the user trajectories in the Geolife dataset. The blue points indicate data points of user trajectories. We divided the region into a hexagonal grid where each cell has the radius of 50 m, which is the service range of a typical Wi-Fi AP [42]. We allocated an edge server to a cell which had been visited by any user, so all users in the dataset can offload computations to the server in the current cell. The wireless network speed was set based on our lab Wi-Fi (50 Mbps for download and 35 Mbps for upload). Edge servers were distributed in the KAIST dataset in the same way.

To simulate realistic user movements, we played back user trajectories in the test sets of our datasets, which were not

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

TABLE III
ACCURACY OF EDGE SERVER PREDICTION (%). THE NUMBER INSIDE PARENTHESIS INDICATES MEAN ABSOLUTE ERROR (M).

Dataset	**Markov**		**SVR**		**RNN**	
	top-1	top-2	top-1	top-2	top-1	top-2
KAIST	4.6	44.4	8.1 (12.9)	54.1 (12.9)	9.2 (12.4)	54.6 (12.4)
Geolife	15.0	32.0	38.1 (31.4)	59.6 (31.4)	36.9 (32.1)	58.1 (32.1)

used for training mobility predictors; the number of mobile users was 31 for KAIST and 138 for Geolife. The location of each user was updated every time interval. All clients used the same DNN model for each simulation run; the DNN model of each client was not shared at the edge server, because in a real scenario the model could be personalized and is likely to be different, thus by default not sharable across different clients. Time to perform DNN partitioning and mobility prediction was ignored, since it is negligible compared to DNN execution time. As we could not prepare hundreds of real edge servers, we profiled the execution time of DNN layers at the server and at the client in advance using *caffe* [41], and performed the simulation using those execution profiles.

2) Accuracy of Mobility Prediction Algorithms: We compared the accuracy of the mobility prediction algorithms (Markov, SVR, RNN) explained in section 3.D. When calculating the accuracy, we only counted *non-futile* predictions, i.e., predictions made just before when a client moves to another server, because futile predictions are useless for proactive migration. For Markov, the top-*n* accuracy means that the prediction is correct when the client visits one of *n* highest probable servers. For SVR and RNN, the prediction is correct when the client visits one of *n* closest servers from the predicted location. We considered top-2 accuracy as well as top-1, because most of top-1 result was the current server, i.e., the client was predicted to stay in the same edge server. Top-2 results always include a server other than the current one, thus more suitable for evaluating the edge server prediction for proactive migration.

Table III summarizes the result. Markov shows definitely lower top-1 and top-2 accuracy than SVR and RNN in both datasets. This is because Markov predictor loses the exact location information of clients when mapping from x, y coordinates to a discrete edge server identifier. SVR and RNN showed similar accuracy and MAE in both datasets. We tested various RNN models, but the best configuration for minimum MAE required just a single LSTM cell (with an output dimension of 32 for KAIST and 16 for Geolife), which is much simpler than RNNs for difficult tasks such as speech recognition [43]. The result implies that location prediction with a short-term trajectory does not require such a complex RNN model, or our training data was not sufficient enough for training. Since linear SVR showed an accuracy similar to RNN and was faster than RNN in terms of both training and testing, we decided to use it in the simulation.

3) Simulation Result: To evaluate the impact of proactive migration on DNN execution performance, we measured the total number of executed DNN queries and hit ratios (the ratio of *hit* cases over the sum of *hit* and *miss* cases) while all clients traverse their trajectories. For baseline (IONN [13]), the clients always incrementally upload DNN layers from scratch whenever connecting to a different edge server (hit ratio = 0%). PerDNN predicts the next location of a client and proactively migrates DNN layers to all edge servers within a certain distance (*r* meters) from the predicted location; edge servers keep those layers for five time intervals (*TTL*=5). *Optimal* is when all DNN layers are always available in all edge servers, so DNN queries are always executed in full speed (hit ratio = 100 %). We only measured the number of queries executed for a time interval right after a client connects to a new edge server, i.e., whenever a cold start occurs, which are our optimization targets.

Fig. 9 shows the result. The hit ratio of the system was 37% (r=50) and 90% (r=100) in KAIST and 43% (r=50) and 70% (r=100) in Geolife; the hit ratio is proportional to the increase of query counts from the *baseline*. Larger *r* means edge servers further away from the predicted location can receive DNN layers, thus increasing the hit ratio. The result of KAIST with r=100 is highly promising, because it means that we can remove 90% of cold starts. In Geolife, the hit ratio was relatively low even with r=100 (70%), because users in Geolife moved much faster than them in KAIST, hindering accurate user location prediction; the average speed of users in KAIST and Geolife was $\sim$0.5 m/s and $\sim$3.9 m/s, respectively. Since Geolife dataset was collected from different modes of transportation, we anticipate that the hit ratio of Geolife can be improved with advanced prediction techniques such as transportation mode inference [37].

MobileNet showed a small increase in query counts, because it is a tiny model that can be quickly uploaded even in the wireless network; the difference between the number of executed queries of *baseline* and that of *optimal*, i.e., *optimizable queries*, was small ($\sim$3.5k). *Inception* and *ResNet* have much more *optimizable queries* than *MobileNet* ($\sim$40k and $\sim$64k, respectively), such that the increase in executed queries was clearly visible. This indicates that PerDNN is more effective for large DNN models with high deployment overheads.

4) Backhaul Traffics: A major cost for proactive migration is the backhaul traffics for migrating DNN layers. We measured the backhaul traffics of each edge server for each time interval in two directions: *uplink traffic*, the sum of all data traffics sent from the server within the time interval, and *downlink traffic*, the sum of all data traffics coming into the server within the time interval. When clients were using *Inception*, the peak uplink/downlink traffic of the most crowded server was 616/205 Mbps in KAIST and 667/359 Mbps in Geolife; at that time, the server was transmitting DNN models simultaneously to 13 clients and 18 clients, respectively. That amount of traffics is beyond the capacity of typical wireless broadband (less than 100 Mbps in average [16]), so expensive wired solutions such as fiber-optic cables

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

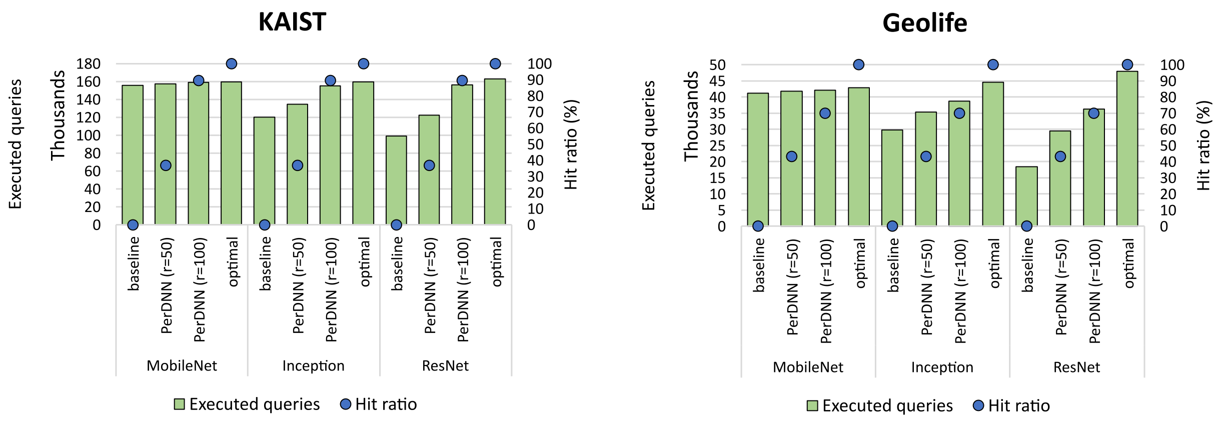


Fig. 9. Number of executed queries and hit ratios during simulation.

would be needed for connecting the crowded server with nearby servers. Fortunately, we found that a relatively small number of servers require such a high traffic (60$\sim$70% of the servers needed less than 100 Mbps uplink/donwnload traffics), so it would be possible to build the PerDNN system on the hybrid network where most edge servers are connected via wireless broadband and some crowded ones are connected via wired links.

5) Fractional Migration: Although we found that the number of crowded servers are relatively small in the previous section, setting up wired connections for those servers would require significant costs. Based on the observation in Section 4.A (DNN execution performance can be highly improved even if we proactively migrate a fraction of a DNN model), we reduced the peak backhaul traffics of crowded servers by migrating only a fraction of a DNN model, instead of the whole model.

We chose the top 5$\sim$7% of the most crowded edge servers (24 servers in KAIST and 86 servers in Geolife) based on the uplink traffics at the peak time and ran a simulation where the chosen servers transmit and receive only a fraction of the server-side layers; the rest of the servers transmit and receive the whole DNN layers as usual. The migrated layers for the crowded servers were selected based on the highest-efficiency-first rule, same as Section 4.A. Fig. 10 shows the result in the *KAIST* dataset. For *Inception*, we could reduce 67% of the peak uplink traffic (616$\rightarrow$206 Mbps) by sacrificing only 2% of the executed queries (when transmitting 43 MB of layers instead of the whole layers). For *ResNet*, the peak uplink traffic decreased by 43% (469$\rightarrow$268 Mbps) in return for 1% reduction of executed queries when transmitting 56 MB of layers. These results imply that the high backhaul traffics of a few crowded servers can be greatly reduced by fractional migration, with little loss in performance.

V. Related Works

A. Computation Offloading

Our study is rooted from the concept of *cyber foraging*, which accelerates mobile applications by offloading clients' computations to nearby surrogates [44]. A popular strategy for cyber foraging is *on-demand customization*, which deploys client's program, encapsulated within a VM, to a server at runtime, so that the server can run the client's custom program [4] [12]. Recently, a more lightweight approach using container for program encapsulation has been proposed due to its fast IO performance and less memory consumption [11]. Most of these studies focus on optimizing the process of VM/container migration to improve the service migration time. On the other hand, this paper takes a different approach that migrates DNN layers between edge servers ahead of time, rather than optimizing the migration process itself.

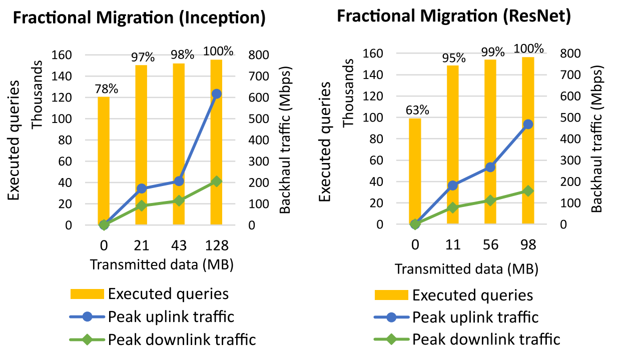


Fig. 10. Impact of fractional migration on peak backhaul traffics and execution performance.

Another important issue on computation offloading is application partitioning. MAUI [45] is an early work on the partitioned execution of application between a client and a server. After MAUI, a tremendous amount of studies on computation partitioning have been piled up [46] [47] [32]. PerDNN can be seen as an extension of cyber foraging with partitioned execution in the context of offloading DNN execution to edge servers.

B. Distributed DNN Inference

Since modern DNN models require substantial computing power, distributed DNN execution across multiple devices

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

has attracted much interest from researchers. Surat Teerapittayanon et al. proposed a DNN model distributed over the cloud, edge, and end devices, to achieve fast and localized DNN inference [48]. MCDNN [49] employs model optimization techniques to offload DNN inferences with maximum accuracy under limited resources (device batteries or cloud costs). These works are different from PerDNN, since they statically divide a DNN model offline [48] or choose a proper model among model variants [49], while PerDNN dynamically partitions a DNN model in a layer level.

To our knowledge, NeuroSurgeon [26] is the first work on the layer-level partitioning of DNN, which finds a partitioning point in a heuristic way. IONN [13] introduced a partitioning algorithm based on shortest path search to find the partitioning of DNN layers. Recently, Chuang Hu et al. proposed a partitioning algorithm applicable to DAG-formed DNNs based on the min-cut algorithm [22]. Unlike the above studies, PerDNN uses runtime GPU information for DNN partitioning to cope with congestion in a server. Also, most of the above studies assume DNN models are already saved at the server (either cloud or edge), not focusing on how to deploy DNN models to offloading servers. Although IONN [13] incrementally transmits DNN layers to reduce the overhead of deploying a DNN model, but, as mentioned in Section 2.B, it does not deal with the cold start issue addressed in this paper. PerDNN complements the above studies, since it helps them to quickly establish optimal offloading status.

C. Other Techniques for Mobility Support

Follow-Me cloud migrates cloud-based services between data centers in a federated cloud, to allow mobile users to always be connected with the optimal data center and data anchor [50]. Follow-Me cloud makes a migration decision based on the distance between the client and the data center, while PerDNN migrates DNN layers based on expected execution latency. Also, Follow-Me cloud performs live migration of the whole VM instance, whereas PerDNN allows migrating a fraction of a DNN model to reduce backhaul traffics.

Our proactive DNN migration resembles the concept of proactive caching (aka femto caching) [19], which reduces data traffic to central clouds by saving popular files at the edge servers in advance [18]. The major difference between proactive caching and PerDNN is that proactive caching saves data at the client far before the client visits the server, but PerDNN transmits DNN layers between edge servers in real time. Also, unlike typical file caching based on user's file popularity, PerDNN selects DNN layers to be offloaded using a partitioning algorithm based on the current runtime states.

VI. Summary and Future Work

This paper explores the feasibility of the public edge server system where mobile users can offload computations of their custom DNN models. We have found an issue regarding the performance degradation at cold start, when DNN models are deployed to a new edge server. PerDNN tackles the issue with proactive migration of DNN layers based on real-time mobility prediction. In the simulation with real world trace datasets and execution profile of real hardware, PerDNN removed 70$\sim$90% of cold starts and achieved 1.6$\sim$2.0x throughput improvement, compared to a baseline with no proactive migration. The backhaul traffics of the system could be sharply reduced with little performance loss, by sending only a small fraction of a DNN model.

This study shows that edge servers can be an effective option for accelerating DNN execution of mobile users in a public place. However, constructing such a system in real world requires many more works. One of the major issues is security. Since our system collects users' recent trajectories for mobility prediction, it can cause the privacy issue, which must be resolved to become available to the public. Also, the exposure of a user's DNN model to public servers has a risk of model inversion attacks [51]. Further investigation and experimentation into these security issues, therefore, is an essential next step. Another important issue is how to handle the offloading of more complex and realistic DNN applications, such as applications simultaneously running multiple DNNs [49] or performing both training and inference on line [10]. Lastly, PerDNN relies on a centralized master server to coordinate the behavior of mobile devices and edge servers. This can cause a single point of failure problem and scalability issues as the number of mobile clients increases. Future work should be carried out to establish a more distributed structure.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1A2B5B02001845). The ICT at Seoul National University provides research facilities for this study.

References

[1] A. Al-Shuwaili and O. Simeone, "Energy-efficient resource allocation for mobile edge computing-based augmented reality applications," *IEEE Wireless Communications Letters*, vol. 6, no. 3, pp. 398–401, 2017.

[2] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 68–81.

[3] K. Wiggers. (2019) Google's stadia uses style transfer ml to manipulate video game environments. [Online]. Available: https://venturebeat.com/2019/03/19/googles-stadia-uses-style-transfer-ml-to-manipulate-video-game-environments/

[4] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, 2009.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[6] F. Lobillo, Z. Becvar, M. A. Puente, P. Mach, F. L. Presti, F. Gambetti, M. Goldhamer, J. Vidal, A. K. Widiawan, and E. Calvanesse, "An architecture for mobile computation offloading on cloud-enabled lte small cells," in *2014 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, April 2014, pp. 1–6.

[7] IEEE. (2019) Ieee 1934. openfog reference architecture. [Online]. Available: https://standards.ieee.org/standard/1934-2018.html

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[9] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, 2015.

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.

[10] R. T. Mullapudi, S. Chen, K. Zhang, D. Ramanan, and K. Fatahalian, "Online model distillation for efficient video inference," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 3573–3582.

[11] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC '17. New York, NY, USA: ACM, 2017, pp. 11:1–11:13.

[12] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013, pp. 153–166.

[13] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 401–411.

[14] Y. Zheng, X. Xie, and W.-Y. Ma, "Geolife: A collaborative social networking service among user, location and trajectory," *IEEE Data(base) Engineering Bulletin*, June 2010.

[15] I. Rhee, M. Shin, S. Hong, K. Lee, S. Kim, and S. Chong, "CRAWDAD dataset ncsu/mobilitymodels (v. 2009-07-23)," Jul. 2009.

[16] SPEEDTEST. (2019) Speedtest global index. [Online]. Available: https://www.speedtest.net/global-index

[17] R. Roy and V. Bommakanti. (2017) Odroid xu4 user manual. [Online]. Available: https://magazine.odroid.com/odroid-xu4

[18] E. Bastug, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5g wireless networks," *IEEE Communications Magazine*, vol. 52, no. 8, pp. 82–89, Aug 2014.

[19] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *IEEE Transactions on Information Theory*, vol. 59, no. 12, pp. 8402–8413, Dec 2013.

[20] M. C. Claudio Cicconetti and A. Passarella, "low-latency distributed computation offloading for pervasive environments," in *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2019, pp. 262–271.

[21] WiGLE. (2018) Wigle wi-fi database. [Online]. Available: https://wigle.net/index

[22] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, 2019, pp. 1423–1431.

[23] K. Tian, Y. Dong, and D. Cowperthwaite, "A full {GPU} virtualization solution with mediated pass-through," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 121–132.

[24] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective {GPU} sharing in {NFV} systems," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 187–200.

[25] NVIDIA. (2018) Nvidia tensorrt inference server. [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/tensorrt-inference-server-guide/docs/index.html

[26] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 615–629.

[27] scikit learn. (2019) Scikit-learn machine learning library. [Online]. Available: https://scikit-learn.org

[28] K.-Y. Shin, H.-J. Jeong, and S.-M. Moon, "Enhanced partitioning of dnn layers for uploading from mobile devices to edge servers," in *Proceedings of the 3rd international workshop on embedded and mobile deep learning*, 2019, pp. 35–40, in press.

[29] R. Jiang, X. Song, Z. Fan, T. Xia, Q. Chen, S. Miyazawa, and R. Shibasaki, "Deepurbanmomentum: An online deep-learning system for short-term urban mobility prediction," in *AAAI*, 2018.

[30] L. Song, D. Kotz, R. Jain, and X. He, "Evaluating location predictors with extensive wi-fi mobility data," in *IEEE INFOCOM 2004*, vol. 2, March 2004, pp. 1414–1424 vol.2.

[31] A. J. Nicholson and B. D. Noble, "Breadcrumbs: Forecasting mobile connectivity," in *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, ser. MobiCom '08. New York, NY, USA: ACM, 2008, pp. 46–57.

[32] L. Yang, J. Cao, S. Tang, D. Han, and N. Suri, "Run time application repartitioning in dynamic mobile cloud environments," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 336–348, July 2016.

[33] D. Ron, Y. Singer, and N. Tishby, "Learning probabilistic automata with variable memory length," in *Proceedings of the seventh annual conference on Computational learning theory*. ACM, 1994, pp. 35–46.

[34] P. Jacquet, W. Szpankowski, and I. Apostol, "A universal predictor based on pattern matching," *IEEE Transactions on Information Theory*, vol. 48, no. 6, pp. 1462–1472, 2002.

[35] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and computing*, vol. 14, no. 3, pp. 199–222, 2004.

[36] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[37] X. Song, H. Kanasugi, and R. Shibasaki, "Deeptransport: Prediction and simulation of human mobility and transportation mode at a citywide level." in *IJCAI*, vol. 16, 2016, pp. 2618–2624.

[38] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social lstm: Human trajectory prediction in crowded spaces," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 961–971.

[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[40] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, 2017.

[41] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[42] Wikipedia. (2018) Long range wi-fi. [Online]. Available: https://en.wikipedia.org/wiki/Long-range˙Wi-Fi

[43] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[44] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal communications*, vol. 8, no. 4, pp. 10–17, 2001.

[45] 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.

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

[47] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Infocom, 2012 Proceedings IEEE*. IEEE, 2012, pp. 945–953.

[48] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 328–339.

[49] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "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, 2016, pp. 123–136.

[50] T. Taleb and A. Ksentini, "Follow me cloud: interworking federated clouds and distributed mobile networks," *IEEE Network*, vol. 27, no. 5, pp. 12–19, 2013.

[51] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1322–1333.

Authorized licensed use limited to: Institute of Software. Downloaded on August 23,2022 at 06:46:00 UTC from IEEE Xplore. Restrictions apply.