# Enhanced Partitioning of DNN Layers for Uploading from Mobile Devices to Edge Servers

Kwang Yong Shin
Seoul National University
Seoul, South Korea
kwangshin@altair.snu.ac.kr

Hyuk-Jin Jeong
Seoul National University
Seoul, South Korea
hjj@altair.snu.ac.kr

Soo-Mook Moon
Seoul National University
Seoul, South Korea
smoon@snu.ac.kr

## ABSTRACT

Offloading computations to servers is a promising method for resource constrained devices to run deep neural network (DNN). It often requires pre-installing DNN models at the server, which is not a valid assumption in an edge server environment where a client can offload to any nearby server, especially when it is on the move. So, the client needs to upload the DNN model on demand, but uploading the entire layers at once can seriously delay the offloading of the DNN queries due to its high overhead. IONN is a technique to partition the layers and upload them incrementally for fast start of offloading [1]. It partitions the DNN layers using the shortest path on a DNN execution graph between the client and the server based on a penalty factor for the uploading overhead. This paper proposes a new partition algorithm based on efficiency, which generates a more fine-grained uploading plan. Experimental results show that the proposed algorithm tangibly improves the query performance during uploading by as much as 55%, with faster execution of initially-raised queries.

## CCS CONCEPTS

• **Human-centered computing** ~ **Mobile computing** • **Computing methodologies** ~ **Distributed computing methodologies** • *Computer systems organization* ~ *Neural networks*

## KEYWORDS

Edge computing; deep neural network; computation offloading; mobile computing

**ACM Reference format:**

Kwang Yong Shin, Hyuk-Jin Jeong and Soo-Mook Moon. 2019. Enhanced Partitioning of DNN Layers for Uploading from Mobile Devices to Edge Servers. In *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications (EMDL'19), June 21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://dx.doi.org/10.1145/3325413.3329788


*EMDL'19, June 21, 2019,* Seoul, Republic of Korea.

ACM ISBN 978-1-4503-6771-4/19/06…$15.00.
DOI: https://dx.doi.org/10.1145/3325413.3329788

## 1. INTRODUCTION

DNN apps are computation intensive, so it is challenging for mobile devices with limited hardware to run them. Current wisdom is to offload the computations to central cloud servers [3] [4] [14], so the servers run the DNNs on behalf of the client using the powerful hardware. This requires the pre-installation of DNN models at the dedicated servers in advance [5] [12].

This requirement is not appropriate for the emerging edge computing environment where the client may send its DNN queries to any nearby generic servers located at the edge of the network [6] [7] [15]. This is especially true when the client is on the move, thus changing the offloading servers frequently. Also, the client may run apps based on its personalized DNN models, existing only on the device. Since the hardware capacity of the edge servers is far limited than that of the centralized servers, it is unreasonable for the edge servers to save many DNN models. Rather, on-demand installation by uploading the client's DNN model to the server would be more practical. A critical issue of the on-demand DNN installation is that the overhead of uploading the whole DNN model takes a long time, resulting in a long delay to use the edge server.

An offloading approach called incremental offloading of neural network (IONN) has been proposed to address this issue [1]. IONN divides a client's DNN model into several partitions of its layers and determines the order of uploading them to the server. The client uploads the partitions to the server one by one, instead of sending the entire DNN model at once. The server incrementally builds the DNN model as each DNN partition arrives, allowing the client to start offloading even partially, before the whole DNN model is uploaded. That is, when there is a DNN query, the server will execute those partitions uploaded so far, while the client will execute the rest of the partitions. This incremental, partial DNN offloading enables mobile clients to use edge servers more quickly, improving the query performance.

To decide the best DNN partitions and the uploading order, IONN uses a heuristic algorithm based on graph data structure, which expresses the collaborative DNN execution between the client and the server. It updates the edge weights according to a

penalty factor and searches for the shortest path. By repeating this process, IONN derives an uploading plan for the DNN partitions that ensures DNN performance converges to the best performance. A problem with the penalty factor approach is that the value is somewhat arbitrary, heuristically decided or differently tuned for each DNN model, and the partition tends to be large, missing the performance opportunity obtainable with a more fine-grained partitioning.

This paper proposes a different partitioning algorithm, based on the efficiency of each partition, the latency reduction benefit divided by the uploading overhead. For the partition chosen in this way, we also try to sub-partition it if the earlier uploading of the sub-partition can accelerate the query execution. The proposed algorithm creates a more fine-grained uploading plan, which can improve the query performance tangibly, by as much as 55% for some model.

The rest of this paper is organized as follows. Section 2 reviews IONN, especially its partitioning algorithm. Section 3 describes the proposed algorithm. Section 4 shows the experimental result. A summary is in Section 5.

## 2. REVIEW OF IONN FOR DNN EDGE

### 2.1 Overview

Generally, a DNN can be viewed as a directed graph whose nodes are layers. Each layer performs its computation on the input data and passes the output data to the next layer. Layers often include parameters that can be trained during the training phase. Once trained, the DNN model can be used for handling the inference query for a given input data.

IONN works as follows. When a client connects to an edge server, it offloads the DNN computations to the edge server. The DNN model is stored on the client device only, and the edge server has no information on the client's DNN model. So, IONN creates an uploading plan using a partitioning algorithm that determines the DNN partitions and the uploading order of each partition. A partition consists of one or multiple layers.

To create an uploading plan, the client needs the profile information on the execution latency of each DNN layer. The client's profile is obtained when the model is installed at the client, which executes the DNN model in advance. The server's profile is estimated from the prediction functions for each type of DNN layer [5], created at the server when installing IONN. The server's profile is delivered to the client at runtime when the client enters its service area.

IONN finds and uploads partitions with a high priority first, and once uploaded, the computations of the layers in those partitions will be done at the server, thus being offloaded. The server will incrementally build the DNN model from the partitions that the client has sent. Knowing which partitions are uploaded and built, the client locally computes the remaining layers that are not yet uploaded. For example, when a DNN query is raised, the computation is performed locally at the client, just before a partition that has been uploaded. Then, the result is transferred to the server, and the server executes the uploaded partitions with it and transfers the new result back to the client. The client continues the query execution, possibly offloading to the server again if deemed advantageous.

DNN layers that would better be at the server to achieve the best expected query performance, should be uploaded as early as possible. However, those layers are not uploaded all at once, but one partition at a time, so that if a query is raised during uploading, the uploaded layers so far will be executed at the server, while local layers will be executed at the client. A partitioning algorithm that partitions the DNN layers considering both the performance benefit and the uploading overhead of each layer is needed. Partitions that are uploaded earlier can have their computations offloaded first, so the priority among the partitions is important.

### 2.2 Original Partitioning Algorithm

IONN builds a graph-based DNN execution model, named NN execution graph, and creates DNN partitions by iteratively finding the fastest execution path on the graph by performing the shortest path algorithm on the graph. Edge weights are adjusted to promote uploading of more layers.

Fig. 1 shows the subgraph corresponding to a single layer, composed of three nodes (1, 2, 3), which will be connected to the next layer, also composed of three layers (4, 5, 6). They depict the client-side execution time (1, 4) as well as the data transfer time to the server (1, 2), the server-side execution time (2, 3), and the data transfer time to the client (3, 4) or the transfer time to the server (3, 5, which is zero). The server-side execution time will also include the upload overhead to initially discourage offloading of layer with high upload overhead.

A path in the graph from the first layer to the last layer depicts a scenario for partitioned execution. IONN finds the shortest path to select the first partition to upload. Those layers in the server-side for the first shortest path will be uploaded and executed at the server initially (initial state). We need to upload more layers to achieve the best performance, the shortest path when the upload overhead is zero in the graph (optimal state); it is not necessarily a path that executes all layers at the server since some layers must still be executed at the client if their data

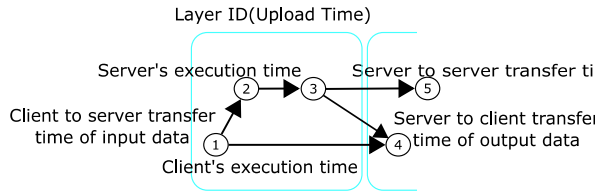


**Figure. 1. Execution time and data transfer time associated with a layer represented as a graph. Edge weights indicate the time taken for each step. Nodes 2, 3, 5 represent that query data is at the server, and 1, 4 represent that query data is at client. Edge (2, 3) and (1, 4) represents execution at server and client, respectively.**

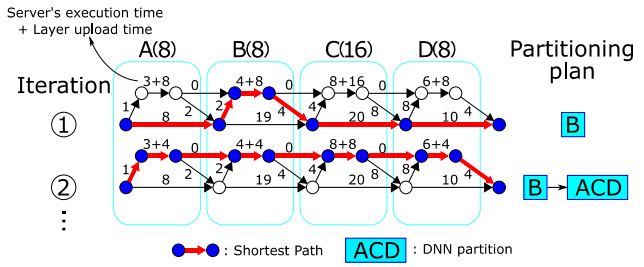


**Figure 2. Illustration of the original partitioning algorithm.**

transfer overhead is too high. IONN iteratively computes the shortest path in the graph after multiplying a penalty factor to the upload overhead in each iteration, which incrementally converts the graph weights to the optimal state. IONN multiplies a constant 0.5 to the penalty factor, thus reducing the upload overhead by half in each iteration. This process repeats until all layers are uploaded or the threshold for the optimal state reaches. The layers in the partition chosen in each iteration will constitute an uploading plan.

Fig. 2 shows an example NN graph composed of four layers (A~D). The upload overhead of each layer is 8, except for C whose upload overhead is 16, and they are depicted next to the server execution time (e.g., 3+8 for A). In the first iteration, the shortest path includes B in the server side. In the second iteration after reducing the upload overhead by half, the new shortest path includes A, C, D in the server side. The uploading plan is uploading [B] first, then [A, C, D].

## 3. ENHANCED PARTITIONING ALGORITHM

In this section, we propose a new partitioning algorithm for IONN based on the efficiency of partitions.

### 3.1 Motivation

IONN used a maximum of eight penalty factors to find partitions: 1, 0.5, 0.25, ..., 0.016, 0.0. When the penalty factor is 1.0, the shortest path finds layers for which uploading leads to a total latency decrease (benefit) higher than its uploading overhead (cost). Reducing the uploading overhead by half in each iteration promotes the uploading of additional layers to eventually reach the optimal state.

There are two issues for the algorithm. The values for the penalty factor are decided somewhat arbitrarily, and may be different for each DNN model. Also, the partitions tend to be large, missing the performance opportunity obtainable with a more fine-grained partitioning, as will be explained shortly.

Fig. 3 (a) shows the overall improvement of the DNN execution latency for the example of Fig. 2, when [B] is uploaded first, followed by [A,C,D]. While [B] is being uploaded for the first 8 time units, there is no latency benefit. After [B] is uploaded at time 8, the total latency is improved by 9 since B's execution at the server reduces its execution time from 19 to 10. This latency benefit is not improved any further until [A,C,D] is uploaded at the time 40, reaching the optimal state, a latency improvement of 31.

However, if IONN chose [A] in iteration 2 and [C,D] in iteration 3, thus making a uploading plan [B]-[A]-[CD], the latency improvement graph will be Fig 3 (b), There is additional latency benefit of 6 during the time 16-40 due to [A] uploaded at the time 16. Uploading of [C,D] reaches the optimal state, as previously.

Finally, if [C] and [D] were uploaded separately, with an uploading plan of [B]-[A]-[C]-[D], there is additional latency benefit of 8 during the time 32-40, as in Fig. 3 (c). However, it should be noted that [B]-[A]-[D]-[C] would achieve a worse latency even than Figure 3 (b), indicating finer-grained partitioning is not always beneficial.

Penalty factor partitioning can find partition [A] with the appropriate penalty factor, but not [C]. If a penalty value between 0.75 and 0.67 was used in the second iteration, [A] would be identified, but there is no clear way of deciding it in advance. Partition [C] cannot be identified by adjusting penalty factor. Instead, [C,D] will always be identified as a single partition.

This simple example indicates there are two improvements

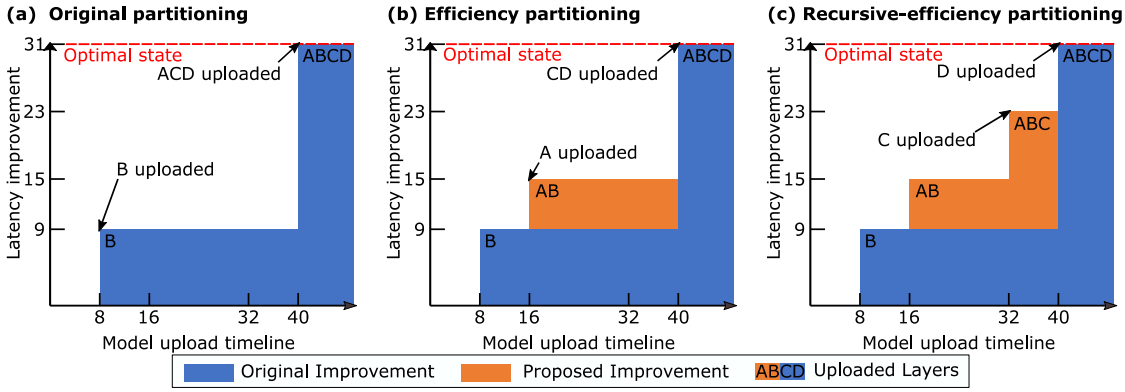


**Fig. 3 Illustration of the incremental latency improvement of different partitioning algorithms**

needed for more fine-grained partitioning. The original algorithm relies on heuristically chosen values for the penalty factors to partition a given DNN model, and the appropriate values cannot be known in advance. There even exist finer partitions that cannot be found by penalty factors with small differences. A new algorithm is proposed that does not rely on heuristically chosen values and finds more fine-grained partitions, while prioritizing partitions with high latency improvement and low upload penalty.

## 3.2 Efficiency-based Partitioning

The main idea is to find the next partition in each iteration using *efficiency*, defined as follows:

Efficiency = latency improvement / upload time

If we define efficiency in the context of penalty-based partition, it would be the first penalty value that can change the shortest path, when we decrease the penalty value from 1.0 repeatedly by a tiny amount. Such a penalty value satisfies:

latency improvement - penalty factor * upload overhead = 0

In other words, efficiency equals to such a penalty factor, but we can decide it deterministically unlike a penalty factor. Based on efficiency, we can devise a greedy algorithm that selects a partition with the highest efficiency repeatedly. We call this approach efficiency-based partitioning in this paper.

The proposed algorithm for the example in Fig. 2 works as in Fig. 4. After [B] is uploaded, we compute efficiency for every possible candidate partition. The partition [A] has an efficiency of 0.75 (since it has an upload overhead of 8 and a latency improvement of 6, as can be seen in Fig. 2), the partition [C,D] has an efficiency of 0.67, and others have a lower efficiency (efficiency of any partition crossing already-uploaded partitions is not calculated, since it is lower than or equal to the best, non-crossing partition). After [A] is uploaded, we compute the efficiency again for the remaining candidate partitions. Among [C], [D], and [C,D], the partition [C,D] has the highest efficiency, so we choose it. This will lead to an uploading plan of [B]-[A]-[C,D].

We need to compute efficiency for every possible sequence of adjacent layers since there are cases where individual layers are inefficient but efficient when grouped into a single partition. When computing the latency improvement, we need to consider the already uploaded layers. That is, depending on whether adjacent layers are already uploaded to the server, the input and output data transfer time between the server and the client can vary significantly for the partition candidates. This means every time a partition is uploaded, efficiencies of the remaining partition candidates may need re-calculating. This favors uploading of a partition that has adjacent layers already uploaded, as the client to server transfer time of input data or the server to client transfer time of output data is replaced by a server to server transfer time, which is zero.

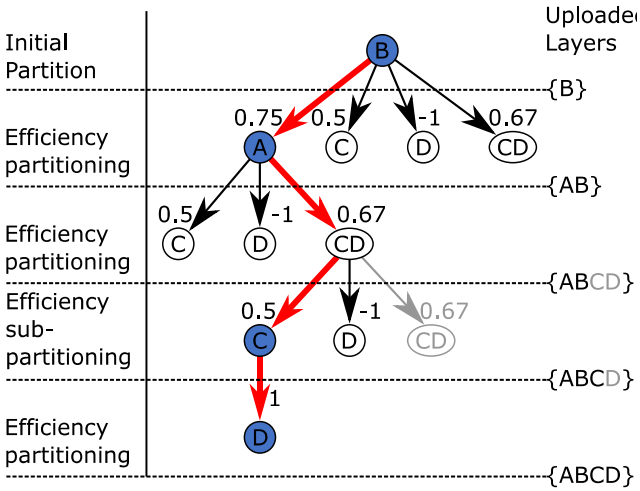


**Figure 4. Illustration of the proposed partitioning algorithm for example in Fig. 2. The generated upload plan is [B]-[A]-[C]-[D].**

## 3.3 Recursive Partitioning

It is desirable for a partition candidate to be recursively sub-partitioned if a sub-partition also improves latency, as in Fig. 3 (c). This approach is called recursive-efficiency-based partitioning in this paper. Fig. 4 illustrates this sub-partitioning, after the partition [C,D] is selected as the most efficient one. Instead of immediately grouping it as a partition, it checks the sub-partitions, [C] and [D], of the partition [C D], to see if a sub-partition with positive efficiency exists. If none, [C D] is grouped as a single partition, but [C] has an efficiency of 0.5. Hence, we select the partition [C] as a separate partition.

If there were multiple sub-partitions with positive efficiency, the sub-partition with highest efficiency is selected. If a partition selected recursively also contains sub-partitions, because it consists of multiple layers, the recursion repeats, until no sub-partition with positive efficiency exists. Then, it means the partition does not contain sub-partition with latency improvement. After selecting partition [C], we return to efficiency-based partitioning in the next iteration, which selects [D]. Because [C] has been uploaded, the efficiency of the partition [D] changes from -1 to 1, higher than the efficiency of [C,D] before recursion. It shows how after selecting a sub-partition by recursion, the remaining sub-partition always has an efficiency higher than that of partition the recursive partitioning was performed on. The final uploading plan is [B]-[A]-[C]-[D].

## 3.4 Overall Proposed Algorithm

The two improvements on the original penalty factor-based partitioning, efficiency-based partitioning and recursive sub-partitioning, are the basis for the final, proposed partitioning algorithm, shown in algorithm 1.

There are two minor optimizations for the algorithm. We can know in advance layers not uploaded in the optimal state, by

computing the shortest path on the NN graph with a penalty factor of zero. Any candidate partition that includes those layers does not need to have its efficiency computed, since it will not be chosen as partition and uploaded. On the other hand, partitions with efficiency is higher than 1 are always beneficial to upload, thus need no further partitioning. It is the initial partition, layers in the server-side of the NN graph with a penalty factor of 1.0

**Algorithm 1** DNN Partitioning Algorithm

**Input:** DNN model description, DNN server and client execution latency (profile or prediction), network speed
**Output:** Partitioning plan

```
1:  procedure PARTITIONING
2:      plan ← []
3:      find optimal layers
4:      find and add initial partition to plan
5:      create partition candidates and calculate their efficiencies
6:      while optimal layer NOT in partitions exist do
7:          p ← candidate with highest efficiency
8:          while sub-partition candidate of p with positive efficiency exist do
9:              p ← sub-partition candidate with highest efficiency
10:         add p to plan
11:         remove partition candidates containing layer in p
12:         update efficiencies
13:     return partitions
```

## 4. PRELIMINARY EVALUATION RESULT

### 4.1 Experiment Environment

We evaluated the proposed partitioning algorithms on the similar IONN environment [1]. We use the caffe [8] framework to implement the DNN offloading system. The client was connected to the server through lab Ethernet, and the experiment was conducted with a network bandwidth limited to 80Mbps by software. The client device is an embedded board Odroid XU4 with ARM big.LITTLE CPU (2.0GHz+ 1.5GHz 4 cores) and 2GB memory [9]. The server has an x86 CPU (3.6GHz 4 cores), GTX 1080 Ti GPU, and 32GB memory.

After the client is connected to the edge server, the client performs the partitioning algorithm and creates an uploading plan. Then, the client starts the incremental uploading of the partitions as well as the execution of the DNN queries. The client repeatedly raises DNN queries, raising a new query right after the previous query finishes, until the entire model is uploaded. Till then, each query is executed collaboratively by the client (local layers) and by the server (uploaded layers), even for the first query, which will wait until the first partition is uploaded completely and then execute (so its execution time is same for the three algorithms; see Fig. 7). Then, it raises five more DNN queries. An average of 10 runs was found for five CNN models: AlexNet [2], Inception [16], ResNet [13], GoogleNet [10] and MobileNet [11].

We experimented with three partitioning algorithms: the original penalty-based algorithm with halving penalty factor values (*original*), the efficiency-based algorithm (*efficiency*), and the efficiency-based algorithm with recursive sub-partitions (*recursive-efficiency*).

### 4.2 Partitioning Behavior

Figure 5 compares the number of partitions for the three algorithms for each model. It also shows the number of partitions uploaded in the first half of the model uploading time and that in the second half separately. The total number of partitions increases tangibly when we move from original to efficiency and recursive-efficiency. The increase in both the first half and the second half for AlexNet and ResNet relates to improved query latencies in both cases, as will be seen later.

Figure 6 shows for each model the relative size of the largest partition compared to the entire model size. Except for Inception, efficiency and recursive-efficiency could divide the largest partition into smaller partitions (the largest partition in Inception was a single layer, thus indivisible).

### 4.3 DNN Query Performance

Figure 7 traces the execution latencies of repeatedly-raised queries for the AlexNet which has a relatively large uploading time. For AlexNet, recursive-efficiency significantly increases the

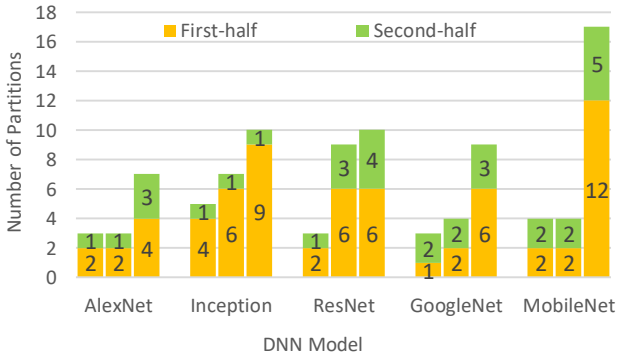


**Figure 5. Number of partitions of original, efficiency, and recursive-efficiency algorithms (from left to right).**

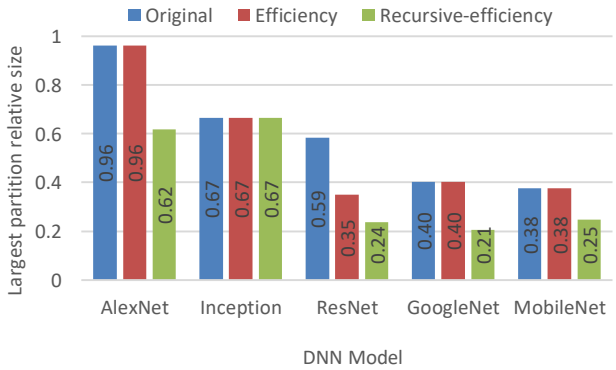


**Figure 6. The relative size of the largest partition, compared to the entire size of the DNN model**

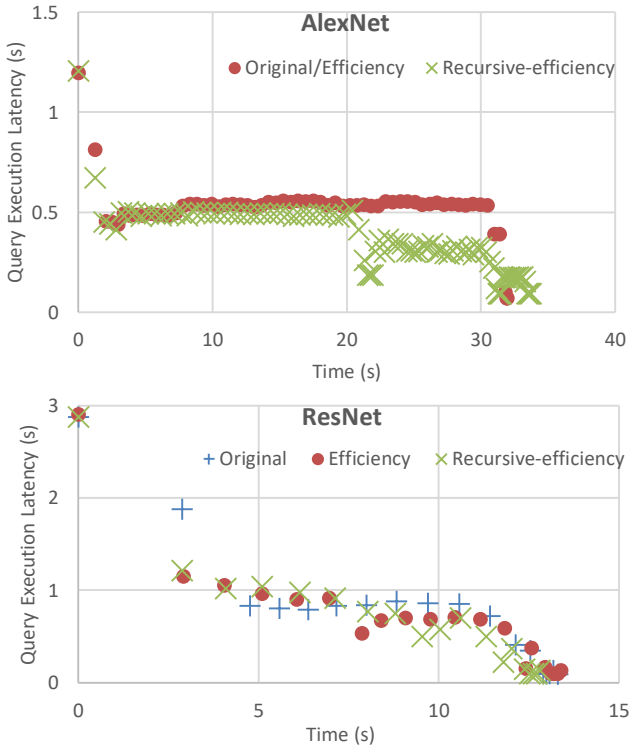


**Figure 7. Query execution latency (shorter is better).**

number of queries executed compared to original, due to the lower latency of query execution, especially during the interval of 22-30 seconds. The number of query processed before the entire model is uploaded increased from 56.8 to 88.1, an increase of 55% (the number of queries per second increased by 48%). These results coincide with the decrease in the relative size of the largest partition in Figure 6 and the increase of the number of partitions in the first and second half of the uploading time in Figure 5.

We also saw some benefit in query latency for ResNet. Original is better in some queries due to implementation issues. For Inception and small-sized models like GoogleNet and MobileNet, there was no clear distinction among three algorithms.

## 4.4 Partitioning Algorithm Overhead

The overhead of the three partitioning algorithms on the client were similar. The speed of the original algorithm is similar to or slightly faster than the efficiency-based algorithm, but both were in the range of 0.1-0.5ms. Modifying the original algorithm to output the same result as efficiency-based one, which was reducing the penalty factor from 1 to 0 by a constant 0.0001, leads to a running time of 100ms. This high overhead is due to its run of the shortest path algorithm around 10,000 times. A small constant was needed because ResNet had partitions identified by 0.0001 difference in the penalty factor. Our proposed algorithm with recursive efficiency runs with a reasonable overhead (0.6ms for AlexNet, for 2ms for MobileNet).

## 5. SUMMARY AND FUTURE WORK

Edge computing is a promising approach to run the DNN models on resource-scarce mobile devices, but requires incremental uploading of layers. This paper proposes an efficiency-based algorithm for partitioning a DNN model to decide the uploading plan of the layers. It prioritizes the partitions with high latency improvement and low uploading overhead, with finer-grained partitions. Experimental results on IONN show that the query performance increases tangibly for some models, while decreasing the initial query completion time, compared to the original penalty-based algorithm.

We need to upgrade the implementation of the server-side DNN, which is expected to improve the result of other models. We also need to evaluate with realistic query raise behaviors of real DNN apps to understand the impact of the proposed algorithm. These are left as a future work.

## ACKNOWLEDGMENTS

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] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. 2018. IONN: Incremental Offloading of Neural Network Computations from Mobile Devices to Edge Servers. In Proceedings of the 2018 ACM Symposium on Cloud Computing (SoCC '18). ACM, New York, NY, USA, 401-411.

[2] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[3] Hewlett Packard Haven, https://www.havenondemand.com/

[4] IBM Alchemy API, https://www.ibm.com/watson/alchemy-api.html

[5] Yiping Kang, et al. (2017). Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 615-629). ACM.

[6] Bonomi, F., Milito, R., Zhu, J., & Addepalli, S. (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).

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

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

[9] ODROID XU4 user manual, https://magazine.odroid.com/odroid-xu4

[10] Szegedy, C. et al. (2015). Going deeper with convolutions. In Proc. of the IEEE conference on computer vision and pattern recognition.

[11] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR abs/1704.04861

[12] L. Yang, J. Cao, H. Cheng and Y. Ji. (2015) "Multi-User Computation Partitioning for Latency Sensitive Mobile Cloud Applications," in *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2253-2266

[13] He, K., et al. (2016). Deep residual learning for image recognition. In Proc of the IEEE conference on computer vision and pattern recognition.

[14] Google Cloud Platform. https://cloud.google.com/compute/pricing

[15] Satyanarayanan, M. (2001). Pervasive computing: Vision and challenges. IEEE Personal communications, 8(4), 10-17.

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