IONN: Incremental Offloading of Neural Network Computations from Mobile Devices to Edge Servers

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

Hyeon-Jae Lee
Seoul National University
Seoul, South Korea
thlhjq@snu.ac.kr

Chang Hyun Shin
Seoul National University
Seoul, South Korea
schyun9212@snu.ac.kr

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

ABSTRACT

Current wisdom to run computation-intensive *deep neural network* (DNN) on resource-constrained mobile devices is allowing the mobile clients to make DNN queries to central cloud servers, where the corresponding DNN models are pre-installed. Unfortunately, this centralized, cloud-based DNN offloading is not appropriate for emerging *decentralized cloud infrastructures* (e.g., cloudlet, edge/fog servers), where the client may send computation requests to any nearby server located at the edge of the network. To use such a generic edge server for DNN execution, the client should first upload its DNN model to the server, yet it can seriously delay query processing due to long uploading time. This paper proposes IONN (Incremental Offloading of Neural Network), a partitioning-based DNN offloading technique for edge computing. IONN divides a client's DNN model into a few partitions and uploads them to the edge server one by one. The server incrementally builds the DNN model as each DNN partition arrives, allowing the client to start offloading partial DNN execution even before the entire DNN model is uploaded. To decide the best DNN partitions and the uploading order, IONN uses a novel graph-based algorithm. Our experiments show that IONN significantly improves query performance in realistic hardware configurations and network conditions.

CCS CONCEPTS

• **Human-centered computing → Mobile computing**; • **Computing methodologies → Distributed computing methodologies**; *Neural networks*;

KEYWORDS

Mobile computing, edge computing, computation offloading, neural network, cyber foraging

ACM Reference Format:
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 SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18)*, 11 pages.
https://doi.org/10.1145/3267809.3267828

1 INTRODUCTION

In recent years, Deep Neural Network (DNN) has shown remarkable achievements in the field of computer vision [22], natural language processing [35], speech recognition [11] and artificial intelligence [34]. Owing to the success of DNN, new applications using DNN are becoming increasingly popular in mobile devices. However, DNN is known to be extremely computation-intensive, such that a mobile device with limited hardware has difficulties in running the DNN computations by itself. Some mobile devices may handle DNN computations with specialized hardware (e.g., GPU, ASIC) [25] [4], but this is not a general option for today's low-powered, compact mobile devices (e.g., wearables or IoT devices).

Current wisdom to run DNN applications on such resource-constrained devices is to *offload* DNN computations to central cloud servers. For example, mobile clients can send their machine learning (ML) queries (requests for execution) to the clouds of commercial ML services [26] [2] [10]. These services often provide servers where pre-trained DNN models or client's DNN models are installed in advance, so that the servers can execute the models on behalf of the client. More recently, there have been research efforts that install the same DNN models at the client as well as at the server, and execute the models partly by the client and partly by the server to trade-off accuracy/resource usage [14] or to improve performance/ energy savings [20]. Both approaches require the pre-installation of DNN models at the *dedicated servers.*

Unfortunately, the previous approaches are not appropriate for the generic use of *decentralized cloud infrastructures* (e.g., cloudlet [33], fog nodes [3], edge servers [32]), where the client can send its ML queries to any nearby generic servers located at the edge of the network (referred to as *cyber foraging* [31]). In this edge computing environment, it is not realistic to pre-install DNN models at the servers for use by the client, since we cannot know which servers will be used at runtime, especially when the client is on the move. 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 DNN model is non-trivial, making the client wait for a long time to use the edge server (see Section 2).

To solve this issue, we propose a new offloading approach, *Incremental Offloading of Neural Network* (IONN). IONN divides a client's DNN model into several partitions 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 of DNN execution even before the entire 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, allowing collaborative execution. This incremental, partial DNN offloading enables mobile clients to use edge servers more quickly, improving the query performance.

As far as we know, IONN is the first work on partitioning-based DNN offloading in the context of cyber foraging. To decide the best DNN partitions and the uploading order, we introduce a novel heuristic algorithm based on graph data structure, which expresses the process of collaborative DNN execution. In the proposed graph, IONN derives the first DNN partition to upload by using a shortest path algorithm, which is expected to get the best query performance initially. To derive the next DNN partition to upload, IONN updates the edge weights of the graph and searches for the new shortest path. By repeating this process, IONN can derive a complete uploading plan for the DNN partitions, which ensures that the DNN query performance increases as more partitions are uploaded to the server and eventually converges to the best performance, expected to achieve with collaborative DNN execution.

We implemented IONN based on *caffe* DNN framework [19]. Experimental results show that IONN promptly improves DNN query performance by offloading partial DNN execution. Also, IONN processes more DNN queries while uploading the DNN model, making the embedded client consume energy more efficiently, compared to the simple all-at-once approach (i.e., uploading the entire DNN model at once).

The rest of this paper is organized as follows. Section 2 illustrates how much overhead is involved in uploading a DNN model for edge computing. In section 3, we briefly review DNN and previous approaches to DNN offloading. In section 4, we explain how IONN works. Section 5 depicts our partitioning algorithm in detail. We evaluate our work in section 6 and show related works in section 7. Finally, we conclude in section 8.

2 MOTIVATION

In this section, we describe a motivating example where the overhead of uploading a DNN model obstructs the use of decentralized cloud servers (throughout this paper, we will refer to the decentralized cloud servers as *edge servers*).

Scenario: A man with poor eyesight wears smart glasses (without powerful GPU) and rides the subway. In the crowded subway station, he can get help from his smart glasses to identify objects around him. Fortunately, edge servers are deployed over the station

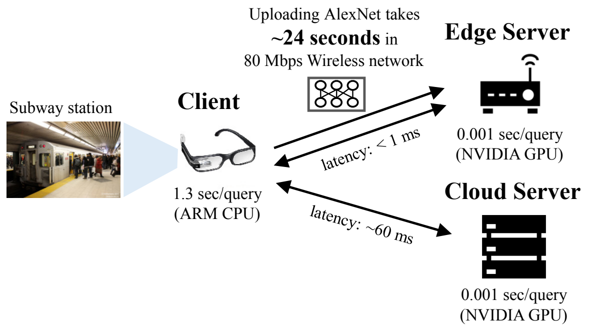


Figure 1: Example scenario of using remote servers to offload DNN computation for image recognition.

(like Wi-Fi Hotspots), so the smart glasses can use them to accelerate the object recognition service by offloading complex DNN computations to a nearby server.

The above scenario is a typical case of *mobile cognitive assistance* [12]. The cognitive assistance on the smart glasses can help the user by whispering the name of objects seen on the camera. For this, it will perform image recognition on the video frames by using DNNs [22] [29]. We performed a quick experiment to check the feasibility of using edge servers for this scenario, based on realistic hardware and network conditions.

Our client device is an embedded board Odroid XU4 [30] with an ARM big.LITTLE CPU (2.0GHz/1.5GHz 4 cores) and 2GB memory. Our edge server has an x86 CPU (3.6GHz 4 cores), GTX 1080 Ti GPU, and 32GB memory. We assumed that the client is connected to Wi-Fi with a strong signal, whose bandwidth is measured to be about 80 Mbps. We experimented with AlexNet [22], a representative DNN for image recognition.

Figure 1 shows the result. Local execution on the smart glasses takes 1.3 seconds to handle one DNN query to recognize an image. Although the CPU on our client board is competitive (the same one used in Samsung Galaxy S5 smartphone), 1.3 seconds per query seems to be barely usable, especially when our smart glasses must recognize several images per second.

If we employ the edge server for offloading DNN queries, one query will take about ~1 ms for execution, which would make a real-time service. However, the DNN model should be available at the edge server in advance to make the edge server ready to execute the queries.

A popular technique to use a random edge server is *VM (virtual machine)-based provisioning*, where a mobile client uploads a service program and its execution environment, encapsulated with VM, to the edge server (or the edge server can download them from the cloud), so that the server can run the service program [32]; some recent studies have proposed using a lightweight container technology instead of VM [23] [27]. If we use these techniques for the purpose of DNN offloading, we would need to upload a VM (or a container) image that includes a DNN model, a DNN framework, and other libraries from the client to the edge server. However, today's commercial DNN framework, such as caffe [19], tensorflow

[8], or pytorch [28], requires a substantial space (more than 3 GB)[1], so it is not realistic to upload such an image on demand at runtime. Rather, it is more reasonable for a VM (or a container) image for the DNN framework to be pre-installed at the edge server in advance, so the client uploads only the client's DNN model to the edge server on demand.

To check the overhead of uploading a DNN model, we measured the time to transmit the DNN model through wireless network. It takes about 24 seconds to upload the AlexNet model, meaning that the smart glasses should execute the queries locally for 24 seconds before using the edge server, thus no improvement in the meantime. Of course, worse network conditions would further increase the uploading time.

If we used a central cloud server with the same hardware where the user's DNN model is installed in advance, we would have obtained the same DNN execution time, yet with a longer network latency. For example, if we access a cloud server in our local region (East Asia) [10], the network latency would be about 60 ms, compared to 1 ms of our edge server due to multi-hop transmission. Also, it is known that the multi-hop transmission to distant cloud datacenters causes high jitters, which may hurt the real-time user experience [32].

Although edge servers are attractive alternatives for running DNN queries, our experimental result indicates that users should wait quite a while to use an edge server due to the time to upload a DNN model. Especially, a highly-mobile user, who can leave the service area of an edge server shortly, will suffer heavily from the problem; if the client moves to another location before it completes uploading its DNN, the client will waste its battery for network transmission but never use the edge server. To solve this issue, we propose IONN, which allows the client to offload partial DNN execution to the server while the DNN model is being uploaded.

3 BACKGROUND

Before explaining IONN, we briefly review a DNN and its variant, Convolutional Neural Network (CNN), typically used for image processing. We also describe some previous approaches to offloading DNN computations to remote servers.

3.1 Deep Neural Network

Deep neural network (DNN) can be viewed as a directed graph whose nodes are *layers*. Each layer in DNN performs its operation on the input matrices and passes the output matrices to the next layer (in other words, each layer is *executed*). Some layers just perform the same operations with fixed parameters, but the others contain *trainable* parameters. The trainable parameters are iteratively updated according to learning algorithms using training data (*training*). After trained, the *DNN model* can be deployed as a file and used to infer outputs for new input data (*inference*). DNN frameworks, such as *caffe* [19], can load a pre-trained DNN from the model file and perform inference for new data by executing the DNN. In this paper, we focus on offloading computations for inference, because training requires much more resources than inference, hence typically performed on powerful cloud datacenters.

A CNN is a DNN that includes convolution layers, widely used to classify an image into one of pre-determined classes. The image classification in the CNN commonly proceeds as follows. When an image is given to the CNN, the CNN extracts *features* from the image using convolution (*conv*) layers and pooling (*pool*) layers. The *conv/pool* layers can be placed in series [22] or in parallel [36] [15]. Using the features, a fully-connected (*fc*) layer calculates the scores of each output class, and a *softmax* layer normalizes the scores. The normalized scores are interpreted as the possibilities of each output class where the input image belongs. There are many other types of layers (e.g., about 50 types of layers are currently implemented in *caffe* [19]), but explaining all of them is beyond the scope of this paper.

3.2 Offloading of DNN Computations

Many cloud providers are offering machine learning (ML) services [26] [2] [10], which perform computation-intensive ML algorithms (including DNN) on behalf of clients. They often provide an application programming interface (API) to app developers so that the developers can implement ML applications using the API. Typically, the API allows a user to make a request (query) for DNN computation by simply sending an input matrix to the service provider's clouds where DNN models are pre-installed. The server in the clouds executes the corresponding DNN model in response to the query and sends the result back to the client. Unfortunately, this centralized, cloud-only approach is not appropriate for our scenario of the *generic* use of edge servers since pre-installing DNN models at the edge servers is not straightforward.

Recent studies have proposed to execute DNN using both the client and the server [20] [14]. *NeuroSurgeon* is the latest work on the collaborative DNN execution using a DNN partitioning scheme [20]. NeuroSurgeon creates a prediction model for DNN, which estimates the execution time and the energy consumption for each layer, by performing regression analysis using the DNN execution profiles. Using the prediction model and the runtime information, NeuroSurgeon dynamically partitions a DNN into the front part and the rear part. The client executes the front part and sends its output matrices to the server. The server runs the rear part with the delivered matrices and sends the new output matrices back to the client. To decide the partitioning point, NeuroSurgeon estimates the expected query execution time for every possible partitioning point and finds the best one. Their experiments show that collaborative DNN execution between the client and the server improves the performance, compared to the server-only approach.

Although collaborative DNN execution in NeuroSurgeon was effective, it is still based on the cloud servers where the DNN model is pre-installed, thus not well suited for our edge computing scenario; it does not upload the DNN model nor its partitioning algorithm considers the uploading overhead. However, collaborative execution gives a useful insight for the DNN edge computing. That is, we can partition the DNN model and upload each partition incrementally, so that the client and the server can execute the partitions collaboratively, even before the whole model is uploaded. Starting from this insight, we designed the incremental offloading of

[1]We measured the size of a docker image for each DNN framework (GPU-version) from dockerhub, which contains all libraries to run the framework as well as the framework itself.

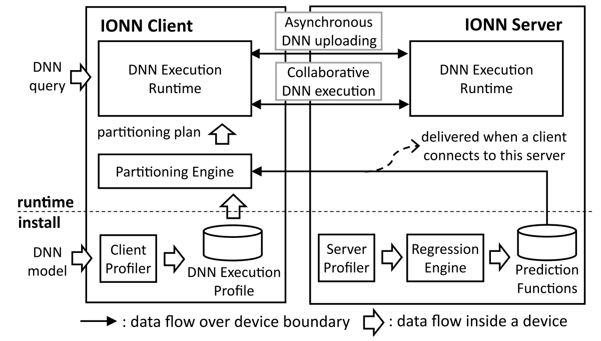


Figure 2: Overall architecture of IONN.

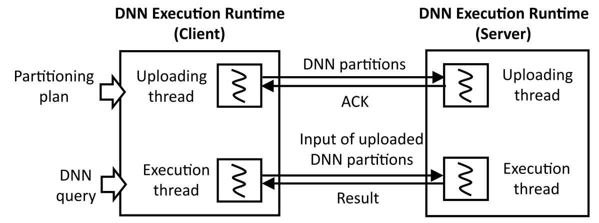


Figure 3: Asynchronous DNN uploading and collaborative DNN execution in DNN Execution Runtime.

DNN execution with a new, more elaborate and flexible partitioning algorithm.

4 IONN FOR DNN EDGE COMPUTING

In this section, we introduce IONN, an offloading system using edge servers for DNN computations. Figure 2 illustrates the overall architecture of IONN, working in two phases. In the *install* phase, IONN collects the execution profiles of DNN layers. In the *runtime* phase, IONN creates an uploading plan that determines the DNN partitions and their uploading order, using the profile information collected in the *install* phase and the dynamic network status. According to the uploading plan, the client asynchronously uploads the DNN partitions to the server using a background thread. When a new DNN query is raised, IONN will execute it collaboratively by the client and by the server, even before the uploading of the partitions completes. We explain both phases more in detail below.

Install Phase (Client) - Whenever a DNN application is installed on the mobile device, *IONN Client* runs the DNN models used in the application and records the execution time of each DNN layer in a file called *DNN execution profile* (lower left of Figure 2). The DNN execution profile will be used by the partitioning engine at runtime to create an uploading plan.

Install Phase (Server) - An edge server cannot know which DNN models to execute in the future, so it is infeasible for the server to collect the DNN execution profiles as the client does. Instead, when installing *IONN Server* on the edge server, we create prediction functions for DNN layers, which can estimate the time for the server to execute a DNN layer according to the type and the parameters of the layer. The prediction functions will be shipped to the client at runtime when the client enters its service area and used for the client to partition a DNN. To create the prediction functions, *IONN Server* performs linear regression on the execution data of DNN layers gathered by running diverse DNN models with different layer parameters (lower right of Figure 2), as NeuroSurgeon does [20]. We used regression functions listed in [20] to estimate the prediction functions. For the layers not mentioned in [20], we performed linear regression using the input size as the model variable.

Runtime phase - Runtime phase starts when a mobile client enters the service area of an edge server. When a client establishes a connection with an edge server, the edge server transmits its prediction functions to the client. Since the size of the prediction functions is small (hundreds of bytes for 11 types of layers in our prototype), the network overhead for sending them is negligible. After the client receives the prediction functions, the *partitioning engine* in the client creates an uploading plan using our graph-based partitioning algorithm (explained in the next section). *DNN execution Runtime* uploads the DNN partitions to the server according to the plan and performs collaborative DNN execution in response to DNN queries. Figure 3 depicts how *DNN Execution Runtime* works in more detail. Since *DNN Execution Runtime* uploads DNN partitions and executes DNN queries concurrently, we need two threads: one for uploading and the other for execution.

The *uploading thread* in the client starts to run as soon as the partitioning engine creates an uploading plan, which is a list of DNN partitions, each of which is composed of DNN layers. First, the uploading thread sends the first DNN partition in the list to the server. The server builds a DNN model with the delivered DNN partition and sends an acknowledgement message (ACK) for the partition back to the client. Then, the uploading thread in the client sends the next partition to the server. This uploading process repeats until the last DNN partition is uploaded to the server.

The *execution thread* executes a DNN query in accordance with the current status of DNN partitions uploaded so far. The client is aware of which partitions have been uploaded to the server by checking if the ACK of each DNN partition arrived. We refer to the partitions currently uploaded to the server as *uploaded* partitions, as opposed to the *local* partitions. When a DNN query is raised, the execution thread executes the *local* partitions until just before the *uploaded* partitions and sends the result (i.e., the input matrices of the *uploaded* partitions) to the server, along with the indices of the DNN layers in the *uploaded* partitions. The server executes DNN layers whose indices are the ones delivered from the client and sends the result (i.e., the output matrices of the *uploaded* partitions) back to the client. The client and the server continue to execute the DNN partitions in this way, until the execution reaches the output layer.

5 DNN PARTITIONING

In this section, we explain how our *partitioning engine* creates an uploading plan. Our partitioning algorithm tries to upload those DNN layers, needed to be at the server to achieve the best expected query performance, as early as possible. However, we do not upload those layers 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 for collaborative execution. For this, the algorithm partitions the DNN layers considering both the performance benefit and the uploading overhead of each layer, so that

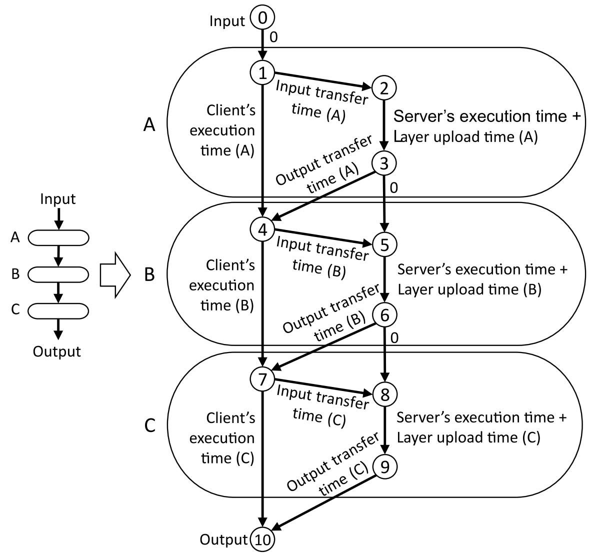

Input
0
1
2
3
4
5
6
7
8
9
Output
10
A
B
C
Client's execution time (A)
Input transfer time (A)
Server's execution time + Layer upload time (A)
Output transfer time (A)
Client's execution time (B)
Input transfer time (B)
Server's execution time + Layer upload time (B)
Output transfer time (B)
Client's execution time (C)
Input transfer time (C)
Server's execution time + Layer upload time (C)
Output transfer time (C)

Figure 4: Example of NN execution graph whose edge weights indicate the execution time of each execution step.

the computation-intensive layers will be uploaded earlier. To derive such an uploading plan, we build a graph-based DNN execution model, named *NN execution graph*, and create DNN partitions by iteratively finding the fastest execution path on the graph.

5.1 Neural Network (NN) Execution Graph

NN execution graph expresses the collaborative execution paths by the client and the server for a DNN query at the layer level. Figure 4 illustrates an NN execution graph created from a DNN composed of three layers. Each layer is converted into three nodes (layer A: 1, 2, 3, layer B: 4, 5, 6, layer C: 7, 8, 9). Nodes in the left side (0, 1, 4, 7, 10) belong to the client, and nodes in the right side (2, 3, 5, 6, 8, 9) belong to the server. Edges between the client nodes (e.g., 1⟶4) indicate local execution, and edges between the server nodes in the same layer (e.g., 2⟶3) indicate server-side execution plus the uploading of the layer. Edges between a client node and a server node (e.g., 1⟶2 or 3⟶4) represent the transmissions of input or output matrices over the network. Each edge is added with a weight to depict the corresponding overhead. Some edges have zero weight (e.g., 0⟶1 or 3⟶5) since no computation or transmission overhead is involved. The client sets up the edge weights as follows. The client can get its local layer execution time from the DNN execution profile and estimate the server's layer execution time using prediction functions. Also, the data and layer transmission time can be calculated by dividing the size of transmitted data by the current network speed.

We can express the execution path of a DNN query as the path on the graph. Let us assume a DNN query is raised with an input data (node 0⟶1). If layer *A* is executed at the client, execution flow will directly go to layer *B* (node 1⟶4). Or if the client offloads the execution of the layer *A* to the server, then execution flow will go to node 2, and then 3. If the next layer (layer *B*) is also executed at the server, execution flow will go from node 3 to node 5. Or, if the layer *B* is executed at the client, execution flow will go to node 4. In this way, we can express the execution path of a DNN query in the graph, and a path from an input (node 0) to an output (node 10) indicates the execution path to run the whole DNN layers. For example, a path 0-1-4-5-6-7-10 in Figure 4 means the client executes the layer *A*, offloads the layer *B*, and continues to execute the layer *C*.

The sum of edge weights on a path from the input to the output indicates the estimated query execution time of the execution path plus the time to upload the DNN layers executed at the server. For instance, the sum of edge weights on the path 0-1-4-5-6-7-10 in Figure 4 is the sum of the time to execute layer *A* and *C* at the client, the time to execute layer *B* at the server, the time to transmit the feature data, and the time to upload the layer *B*. So, if we compute the shortest path on the NN execution graph, the path will tell which layers should be initially uploaded to the server to minimize query execution time.

5.2 Partitioning Algorithm

Our DNN partitioning problem is to decide which layers to include in the uploading partitions, and in what order to upload them, to minimize the query execution time. Unfortunately, it is impossible to find an optimal solution unless we fully know the future occurrence of queries; we need to know how soon the next query will occur to decide an optimal amount of DNN partitions to upload now (e.g., if the next query comes late, we would better upload a large partition, but if it comes soon, we would better upload a small partition quickly). Since it is hard to predict the client's future query pattern, we propose a heuristic algorithm that can work irrespective of the pattern, based on two rules. First, we prefer uploading DNN layers whose performance benefit is high and whose uploading overhead is low. This will make the server quickly build a partial DNN with high expected speedup, thus improving the query performance rather early. Second, we do not send unnecessary DNN layers, those that do not result in any performance increase, to reduce the cost associated in offloading them.

Our algorithm is based on the shortest path in the NN execution graph. As mentioned above, the shortest path on the NN execution graph represents the fastest execution path for a DNN query in the *initial state* (i.e., no DNN layers are uploaded). On the other hand, if we set the layer upload time of all DNN layers in the graph to zero, the graph will represent the situation where the whole DNN model is uploaded to the server, which is the *optimal state* for offloading DNN execution. If we compute the shortest path of such a graph, it will be the execution path with the best query performance we can achieve with collaborative execution; it is not necessarily a path that executes all layers at the server (i.e., offloading the entire DNN) since some layers might better be executed at the client due to the high data transmission overhead. To eventually reach the best performance, we create an uploading plan by iteratively computing the shortest path in the NN execution graph, while changing the edge weights of the graph from the *initial state* to the *optimal state*.

Algorithm 1 describes our DNN partitioning algorithm that computes an uploading plan, a list of DNN partitions named *partitions*, which is an empty list initially (line 2). We first build an NN execution graph using the DNN model description, the DNN execution

Algorithm 1 DNN Partitioning Algorithm

```
Input: DNN model description, DNN execution profile, prediction functions, network speed, $K$ (positive number less than 1)
Output: Uploading plan (a list of DNN partitions)
1:  procedure PARTITIONING
2:      partitions ← [ ] ;
3:      $n$ ← 0;
4:      Create NN execution graph using input parameters;
5:      while $K^n \geq 0.01$ do        ▷ Until layer upload time becomes ≈ 0
6:          Search for the shortest path in the NN execution graph;
7:          Create a DNN partition and add it to partitions;
8:          Update the edge weights of the NN execution graph by multiplying $K$ to layer upload time;
9:          $n \leftarrow n + 1$;
10:     return partitions
```

profile, the prediction functions, and the current network speed (line 4). Next, we search for the shortest path from the input to the output in the NN execution graph (line 6). We identify those layers whose server-side nodes are included in the shortest path, create a DNN partition composed of those layers, and add it to *partitions* (line 7). Since the shortest path is computed based on the edge weights including both the layer execution time and the layer upload time, the DNN partition would include DNN layers with high expected speedup but short upload time, satisfying our first rule. We can use a shortest path algorithm for DAG using topological sorting, whose time complexity is O(n) (n : the number of layers) [9]. Next, we reduce the layer upload time by multiplying K (positive number less than 1) (line 8) and search for the new shortest path in the updated graph (line 6). The new shortest path is likely to include more server-side nodes than the previous one due to the reduction of the server-side edge weights. We create the next DNN partition with the layers whose server-side nodes are newly included in the new shortest path. We repeat this process until the edge weights for the layer upload time become almost zero (line 5), which is nearly the *optimal state* for offloading. Hence, the query performance becomes the best performance after the last partition is uploaded, satisfying our second rule. The output of the algorithm is a list of DNN partitions, and the client will upload the partitions in the list from the first to the last.

Figure 5 illustrates our partitioning algorithm with an example DNN composed of four layers (A~D). We first build an NN execution graph for the initial state as explained in section 5.1. In the first iteration, we search for the shortest path from an input to an output in the graph (depicted in red arrows) and create a DNN partition [B] since only the layer B is in the server-side. Next, we multiply K (0.5 in this example) to the layer upload time and search for the new shortest path in the next iteration. We create the second DNN partition [A,C], since the server-side nodes of A and C are newly included in the shortest path. In the third iteration, the shortest path is the same as before although the layer upload time is reduced by half, thus no DNN partition is newly created. In fact, layer D will never be uploaded, because the shortest path will not include

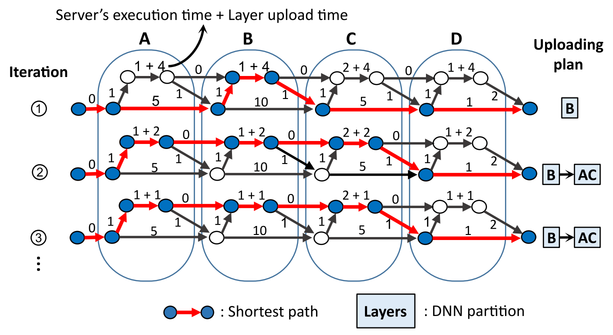


Figure 5: Illustration of our DNN partitioning algorithm.

the server-side nodes of the layer D even when the layer upload time reaches zero. This means the layer D would better be executed at the client for the best query performance. So, our algorithm generates an uploading plan, *partitions*=[[B], [A,C]]. The uploading thread will upload the layer B first, then the layers A and C. In this way, we can quickly upload a computation-intensive layer (B) and ultimately achieve the best query performance by uploading three layers (A, B, C) as needed. We could also save the client's energy consumption by not uploading the layer D.

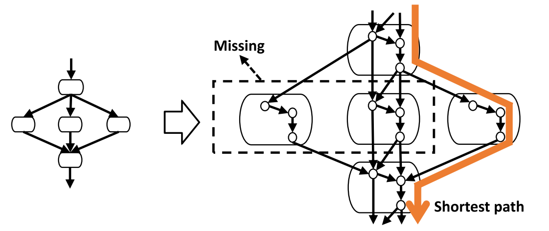
We can adjust the granularity of DNN partitions by changing the value of K. If the value of K is small, the weight for the layer upload time will decrease sharply in each iteration. This will let the partitioning algorithm finish within a small number of iterations, therefore making a few, large DNN partitions. On the other hand, a large K will lead to many iterations and create many, small DNN partitions. We evaluate the impact of K values in Section 6.

Note that our algorithm assumes that edge servers have plenty of network/computing resources, so contention for shared edge server resources between multiple clients is negligible. The partitioning algorithm with multiple clients under limited edge server resources is left for future work.

5.3 Handling DNNs with Multiple Paths

There is an issue in our partitioning algorithm to handle DNNs with multiple paths. Figure 6 (a) illustrates the problematic situation where our algorithm does not work. The example NN has a layer whose output is delivered to three layers, and the outputs of the three layers are concatenated and given as the input of the next layer (left in Figure 6 (a)). If we convert the layers one by one as we did in Figure 4, then the NN execution graph will be created as the right of Figure 6 (a). In this graph, the shortest path from an input to an output will include just one layer among the three layers in the middle, missing the execution of the rest two layers. This will derive a wrong uploading plan based on incomplete execution path.

To solve this problem, we build NN execution graph as if the original NN does not have multiple paths, as shown in Figure 6 (b). First, we find *dominators* of the output layer, which are layers that must be included in the path from the input to the output [1]. Next, we build NN execution graph as if layers in between two neighboring dominators (layers between A and B) and the latter dominator (B) are just one layer. The edge weights of the combined layers are shown in the right side of Figure 6 (b). Since



(a) Problematic conversion of a DNN with multiple paths (some edges are omitted)

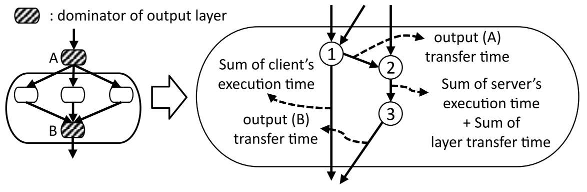

: dominator of output layer
A
B
Sum of client's execution time
output (B) transfer time
output (A) transfer time
Sum of server's execution time + Sum of layer transfer time
1
2
3
(b) Building NN execution graph as if there is no multiple paths

Figure 6: (a): Conversion of a DNN with multiple paths. (b): Building NN execution graph as if the DNN does not have multiple paths.

Name	Size (MB)	Number of layers	Reference
AlexNet	233	24	[22]
Inception	129	312	[37]
ResNet	98	245	[15]
GoogleNet	27	152	[36]
MobileNet	16	110	[16]

Table 1: DNNs For Evaluation

Name	All_at_once	IONN (K=0.1)	IONN (K=0.5)
AlexNet	28.7	30.3	30.7
Inception	16.4	16.7	16.8
ResNet	12.1	12.6	13.0
GoogleNet	3.9	3.9	4.4
MobileNet	2.3	2.5	2.8

Table 2: Uploading Completion Time (second)

there are no paths bypassing the dominators (due to the definition of dominators), the path from the input to the output will not miss any layer execution.

6 EVALUATION

In this section, we evaluate IONN in terms of the query performance and the mobile device's energy consumption.

6.1 Experimental Environment

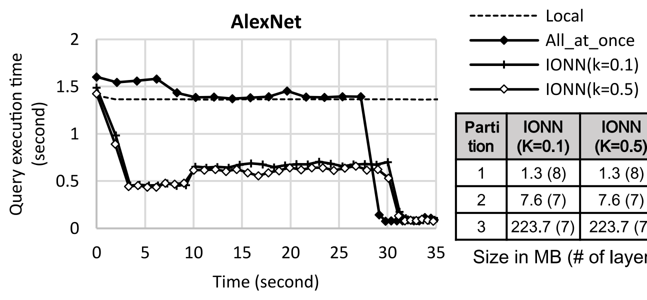
We implemented IONN on *caffe* DNN framework [19] using a network library *boost.asio*. We used the same client device and the edge server as those used in the experiment in Section 2. The client was connected to our lab Wi-Fi which has a bandwidth of about 80 Mbps. The server was connected to the internet with Ethernet. We made a cognitive assistance scenario similar to the example in Section 2. We assumed the client repeatedly raises a DNN query for image classification after pre-processing an incoming video frame for 0.5 second, i.e., pre-processing (0.5 sec) → DNN execution → pre-processing (0.5 sec) → DNN execution →... We experimented with five CNNs ranging from a small DNN (MobileNet) used in mobile devices to larger DNNs. Table 1 shows the size of DNN models and the number of layers after each DNN model is loaded on the *caffe* framework. We compared IONN with the *local* execution, and the *all-at-once* execution where the client uploads the entire DNN model and then offloads DNN queries to the server; DNN queries raised during uploading are executed at the client.

6.2 DNN Query Performance

Figure 7 shows the execution time of DNN queries in three cases: *IONN (K=0.1)*, *IONN (K=0.5)*, *all_at_once*, compared to *local*. The X value of each data point is the time when a DNN query is raised. Y value is the time spent to execute the DNN query. It should be noted that the number of executed queries (the number of data points) differs for each case; more queries are executed if the query execution time is shorter. *All_at_once* starts to offload the DNN execution only after the whole DNN is uploaded, so its query performance is low (same as *local*) until the uploading is over. On the other hand, *IONN* (both *K=0.5* and *K=0.1*) offloads partial DNN execution before the uploading is over, so the query performance is much better while uploading the DNN model. Also, we observed the query execution time of both *IONN*s rapidly decreases after a few queries and eventually reaches the minimal, which is the same execution time of *all_at_once* when the uploading is over. This implies that *IONN* can quickly offload computation-intensive layers and eventually achieve the best performance.

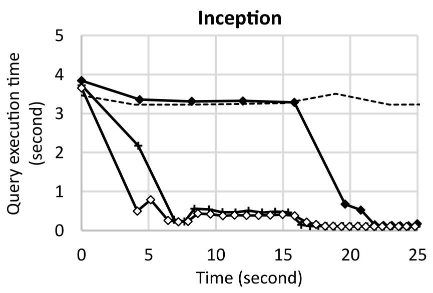
Figure 7 also shows the impact of the K value on the granularity of DNN partitions (i.e., number/size of DNN partitions) and the query performance. As expected, *IONN (K=0.5)* created more partitions than *IONN (K=0.1)* (except for AlexNet), so its average size of the partition was smaller. Since smaller DNN partitions can be uploaded to the server more quickly, the query performance of *IONN (K=0.5)* will improve earlier than *IONN (K=0.1)*. This is the reason why *K=0.5* performed better than *K=0.1* in the 3rd and 4th queries of ResNet; the large second partition of *K=0.1* was still being uploaded, while the small second partition of *K=0.5* had been already uploaded. A similar result can be observed in the 2nd query of Inception, GoogleNet, and MobileNet.

Another expected impact of K value is network overhead to handle multiple DNN partitions. The larger K value will create more, smaller partitions, and the total time to upload a DNN model will increase due to the handling of more ACK messages. Table 2 shows the impact of the network overhead on the time to upload the whole DNN layers. As expected, the uploading completion time of each DNN model is longer when the number of DNN partitions is larger (*All_at_once* < *IONN (K=0.1)* ≤ *IONN (K=0.5)*). But the difference of the uploading completion time is small, meaning that the overhead of uploading multiple partitions is insignificant.



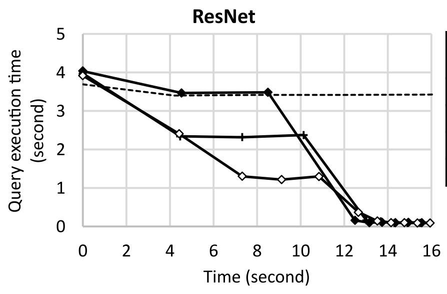
Partition	IONN (K=0.1)	IONN (K=0.5)
1	1.3 (8)	1.3 (8)
2	7.6 (7)	7.6 (7)
3	223.7 (7)	223.7 (7)

Size in MB (# of layers)



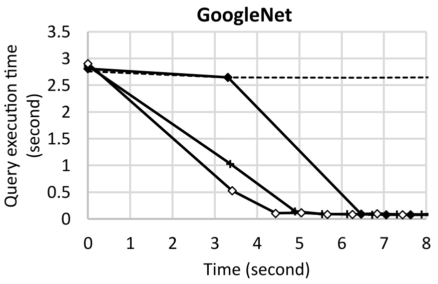
Partition	IONN (K=0.1)	IONN (K=0.5)
1	5.5 (76)	5.5 (76)
2	22.3 (233)	3.3 (23)
3	85.4 (1)	11.9 (124)
4	-	5.4 (23)
5	-	1.7 (63)
6	-	85.4 (1)

Size in MB (# of layers)



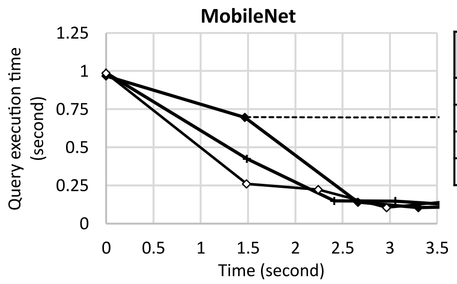
Partition	IONN (K=0.1)	IONN (K=0.5)
1	5.6 (108)	5.6 (108)
2	84.3 (134)	27.2 (87)
3	7.8 (1)	57.2 (47)
4	-	7.8 (1)

Size in MB (# of layers)



Partition	IONN (K=0.1)	IONN (K=0.5)
1	2.6 (41)	2.6 (41)
2	20.2 (107)	10.7 (76)
3	3.9 (2)	9.5 (31)
4	-	3.9 (2)

Size in MB (# of layers)



Partition	IONN (K=0.1)	IONN (K=0.5)
1	0.5 (45)	0.5 (45)
2	11.8 (64)	5.7 (48)
3	3.9 (1)	6.1 (16)
4	-	3.9 (1)

Size in MB (# of layers)

Figure 7: Execution time of DNN queries and the size of each DNN partition in our benchmark DNNs.

We observed that the query execution time in *IONN* sometimes increases during the uploading of the DNN layers, as at 10~30 second in AlexNet and 8~16 second in Inception. This phenomenon appears to be due to the transmission of the last DNN partition, which is much larger than other partitions, interfering severely with the transmission of feature data. Nonetheless, the performance benefit of offloading is even higher than the interference overhead, so the overall query execution time of *IONN* is much shorter than that of *all_at_once*.

6.3 Accuracy of Prediction Functions

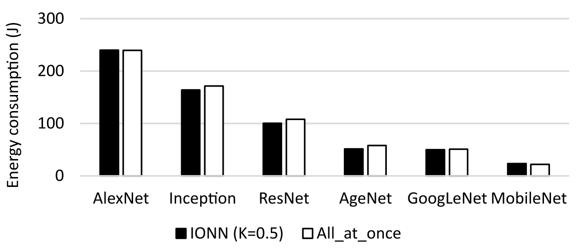
Our uploading plan is created using prediction functions for the server’s layer execution time, so the accuracy of the prediction functions will affect the preciseness of the edge weights in the NN execution graph, thus the final uploading plan. We evaluated the accuracy of the prediction functions as follows. We compared the uploading plan generated from our *predicted* layer execution time, with the uploading plan generated from the *real* layer execution time gathered by recording the time on the server. The uploading plans from both configurations were the same; the number of partitions and the layers included in each partition were the same. This means that our prediction functions are good enough for IONN to generate an accurate uploading plan under real hardware and network conditions.

For a statistical analysis, we calculated the coefficient of determination $(R^2)^2$ and the root mean square error (RMSE) of the prediction functions after the regression. Table 3 shows the result. The R^2 values of all layers, except the conv layer, are close to 1, which means that the prediction functions are suitable for estimating the layer execution time. Although the prediction function for the conv layer has low R^2 value (0.428), the final uploading plan generated by using the prediction function was the same as the one using the real data. This is because the server’s execution time is much smaller than the client’s execution time, so the error of the predicted server’s execution time has little effect on the final uploading plan. The RMSE in Table 3 backs up the statement. The RMSE of the conv layer is 0.025 ms, which is negligible compared to the client’s conv layer execution time (about tens of milliseconds), so the final uploading plan would be hardly affected by the error of the predicted server’s execution time.

[2]The coefficient of determination is the proportion of variation in the dependent variable which is explained by the independent variables. This value can be used to measure the accuracy of a prediction model. R^2 can take 0 as minimum (bad model), and 1 as maximum (good model).

Layer Type	R^2	RMSE (ms)	Layer Type	R^2	RMSE (ms)
Conv	0.428	0.025	FC	0.997	1.291
ReLU	0.999	0.001	Softmax	1.000	0.256
Pooling	0.853	0.002	BatchNorm	0.953	0.004
LRN	1.000	0.009	Scale	0.953	0.002
Concat	1.000	0.018	Eltwise	0.991	0.002

Table 3: R^2 and RMSE of Prediction Functions



Executed queries	AlexNet	Inception	ResNet	AgeNet	GoogLeNet	MobileNet
IONN	31	26	6	11	7	3
All_at_once	20	9	5	8	3	3

Figure 8: Execution time of DNN queries and the size of each DNN partition in our benchmark DNNs.

6.4 Energy Consumption

We measured the energy consumption of our client board using SmartPower2 [30] until the client finishes uploading its DNN model to the server (e.g., 0~30.5 seconds for *IONN* in AlexNet in Figure 7). Figure 8 shows the result. In all benchmarks, *IONN* and *all_at_once* consumed a similar amount of energy (overall, *IONN* consumed slightly less energy than *all_at_once*, but the difference is insignificant). This implies the overhead of incremental offloading (e.g., ACK messages, a longer uploading time) is not burdensome for mobile devices. Figure 8 also shows the number of queries executed during the uploading. *IONN* executed more queries than *all_at_once* in all benchmarks except MobileNet, showing higher throughput. These results imply that *IONN* improves query performance without increasing energy consumption compared to *all_at_once*.

7 RELATED WORK

IONN is built upon previous studies in computation offloading, especially based on *cyber foraging*, an offloading technique where a mobile device offloads computations to nearby servers [31]. *Cloudlet* is a decentralized cloud server proposed to realize the cyber foraging with cloud infrastructure [33]. A widely-used solution for leveraging the cloudlet is to customize the cloudlet at runtime according to the client's purpose by using VM (Virtual Machine) technology [33]. A mobile client sends a VM image, where back-end software is installed, to the cloudlet. The cloudlet creates a VM instance from the VM image and lets the instance serve the front-end software in the client. Our approach focuses on offloading DNN execution and the overhead of uploading a DNN model, hence orthogonal to the VM-based customization. In fact, both IONN and the customization scheme can be used simultaneously (e.g., we can customize an edge server with a VM image that contains IONN software and then perform incremental offloading).

Extensive researches have been performed on computation offloading using a partitioning scheme. MAUI [6] partitions the application execution using static program analysis as well as dynamic profiling, which is a method widely used by other researchers [21] [5]. Lei Yang et al. [39] proposed a partitioning algorithm for reducing the latency of mobile cloud applications under multi-user environments. They model an application as a sequence of modules and determines where to execute each module (client or server) by solving a recursive formula. The shortest path algorithm used in IONN is similar to the recursive algorithm proposed in [39], although IONN treats DNN structure which is more complex than the sequence of modules, requiring handling of multiple paths (section 5.3). Also, [39] does not consider the uploading overhead.

Recently, more DNN-focused offloading approaches have been studied. MCDNN offloads DNN execution for streaming data in multi-programming environments [14]. MCDNN creates variants of a DNN model and chooses DNN models among them for a given task to satisfy resource/cost constraints with maximal accuracy. MCDNN assumes the same DNN models are pre-installed at the client and the server, which is different from our edge scenario that uploads DNN models at runtime. Also, MCDNN does not focus on DNN partitioning.

As far as we know, NeuroSurgeon explained in Section 3.B is the first work on DNN partitioning [20]. However, it allows only fixed, two-way partitioning (front layers by the client and rear layers by the server), while IONN can make a more flexible partitioning (e.g., front layers by the client, middle by the server, and rear by the client), depending on the computation power of server/client and the transmission overhead of layers/feature data. As MCDNN, NeuroSurgeon also requires the server to store the DNN model in advance, which is different from IONN.

Machine learning researchers are actively developing techniques that reduce the amount of computation and the size of DNN models to run DNNs on mobile devices. MobileNet [16] decreases the amount of convolution computation by replacing point-wise convolution to depth-wise separable convolution. SqueezeNet [18] is a small DNN (~4.8 MB) designed to have few model parameters with an accuracy similar to AlexNet (~233 MB). Applied with DNN compression techniques [13], the size of SqueezeNet drops to 0.47 MB [18]. If a mobile user wants to offload the execution of such a tiny DNN, the benefit of IONN would be insignificant, because the whole DNN model might be uploaded soon. However, DNNs for complex tasks still have complicated structure and a large model size. For example, SENet [17], one of the winners of ILSVRC 2017, consists of more than 900 layers and has a model size of ~440 MB. Also, emerging end-to-end DNN architecture, which performs an entire process to solve a cognitive [24] or generative task [7] [38] in a single DNN, might lead to the increase of the DNN model size. It would be difficult to run such a large DNN model on mobile devices. IONN is a feasible solution for offloading large, complex DNNs in mobile applications, because incremental offloading results in high performance benefits even during the uploading stage of DNN models.

8 SUMMARY AND FUTURE WORK

This paper proposes IONN, a novel DNN offloading technique for edge computing. IONN partitions the DNN layers and incrementally uploads the partitions to allow collaborative execution by the client and the edge server even before the entire DNN model is uploaded. Experimental results show that IONN improves both the query performance and the energy consumption during DNN model uploading, compared to a simple all-at-once approach.

Using IONN, we can build DNN-centric edge servers where a mobile client can offload its custom DNN computations without a long waiting time. Since DNN is the de-facto standard for many cognitive tasks (e.g., object detection, speech recognition) these days, we believe DNN-centric edge servers have a potential to become the frontier of cloud computing infrastructure to handle cognitive computations of end devices (e.g., IoT, wearables). To make our idea more practical, we will extend IONN in several directions. First, we will explore a more realistic situation where a client moves around multiple edge servers and freely changes the servers. In such a case, the seamless handoff of DNN services would be crucial for user experience. Also, we will study an advanced DNN partitioning algorithm to handle the case where multiple mobile clients simultaneously offload DNN execution to edge servers. Finally, we plan to extend IONN to support more diverse DNNs, (e.g., RNN or CRNN) and other DNN computations (e.g., DNN training) as well.

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] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, Principles, Techniques. *Addison Wesley* 7, 8 (1986), 9.

[2] Alchemy API. 2009. IBM Alchemy API. (2009). Retrieved April 23, 2018 from https://www.ibm.com/watson/alchemy-api.html

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. 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, 13–16.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[5] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems.* ACM, 301–314.

[6] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services.* ACM, 49–62.

[7] Pengfei Dou, Shishir K. Shah, and Ioannis A. Kakadiaris. 2017. End-to-end 3D face reconstruction with deep neural networks. *CoRR* abs/1704.05020 (2017). arXiv:1704.05020 http://arxiv.org/abs/1704.05020

[8] Martín Abadi et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). arXiv:1603.04467 http://arxiv.org/abs/1603.04467

[9] Michael T. Goodrich. 1996. ICS 161: Design and Analysis of Algorithms Lecture notes. (1996). Retrieved April 23, 2018 from https://www.ics.uci.edu/~eppstein/161/960208.html

[10] Google. 2008. Google Cloud Platform. (2008). Retrieved April 23, 2018 from https://cloud.google.com

[11] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on.* IEEE, 6645–6649.

[12] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services.* ACM, 68–81.

[13] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). arXiv:1510.00149 http://arxiv.org/abs/1510.00149

[14] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. 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, 123–136.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[16] 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 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[17] Jie Hu, Li Shen, and Gang Sun. 2017. Squeeze-and-Excitation Networks. *CoRR* abs/1709.01507 (2017). arXiv:1709.01507 http://arxiv.org/abs/1709.01507

[18] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia.* ACM, 675–678.

[20] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. 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, 615–629.

[21] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE.* IEEE, 945–953.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12).* Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[23] Andrew Machen, Shiqiang Wang, Kin K. Leung, Bong Jun Ko, and Theodoros Salonidis. 2018. Live Service Migration in Mobile Edge Clouds. *Wireless Commun.* 25, 1 (Feb. 2018), 140–147. https://doi.org/10.1109/MWC.2017.1700011

[24] Yajie Miao, Mohammad Gowayyed, and Florian Metze. 2015. EESEN: End-to-End Speech Recognition using Deep RNN Models and WFST-based Decoding. *CoRR* abs/1507.08240 (2015). arXiv:1507.08240 http://arxiv.org/abs/1507.08240

[25] NVIDIA. 2008. NVIDIA Tegra. (2008). Retrieved April 23, 2018 from https://www.nvidia.com/object/tegra.html

[26] Hewlett Packard. 2013. HP Haven. (2013). Retrieved April 23, 2018 from https://www.havenondemand.com

[27] C. Pahl and B. Lee. 2015. Containers and Clusters for Edge Cloud Architectures – A Technology Review. In *2015 3rd International Conference on Future Internet of Things and Cloud.* 379–386. https://doi.org/10.1109/FiCloud.2015.35

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[29] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 779–788.

[30] Rob Roy and Venkat Bommakanti. 2017. ODROID XU4 user manual. (2017). Retrieved April 23, 2018 from https://magazine.odroid.com/odroid-xu4

[31] Mahadev Satyanarayanan. 2001. Pervasive computing: Vision and challenges. *IEEE Personal communications* 8, 4 (2001), 10–17.

[32] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39.

[33] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009).

[34] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[35] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14).* MIT Press, Cambridge, MA, USA, 3104–3112. http://dl.acm.org/citation.cfm?id=2969033.2969173

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. 2015. Going deeper with convolutions. Cvpr.

[37] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR* abs/1512.00567 (2015). arXiv:1512.00567 http://arxiv.org/abs/1512.00567

[38] Anh Tuan Tran, Tal Hassner, Iacopo Masi, and Gérard G. Medioni. 2016. Regressing Robust and Discriminative 3D Morphable Models with a very Deep Neural Network. *CoRR* abs/1612.04904 (2016). arXiv:1612.04904 http://arxiv.org/abs/1612.04904

[39] Lei Yang, Jiannong Cao, Hui Cheng, and Yusheng Ji. 2015. Multi-user computation partitioning for latency sensitive mobile cloud applications. *IEEE Trans. Comput.* 64, 8 (2015), 2253–2266.