JointDNN: An Efficient Training and Inference Engine for Intelligent Mobile Cloud Computing Services

Amir Erfan Eshratifar
Dept. of Electrical Engineering
University of Southern California
Los Angeles, California
eshratif@usc.edu

Mohammad Saeed Abrishami
Dept. of Electrical Engineering
University of Southern California
Los Angeles, California
abri442@usc.edu

Massoud Pedram
Dept. of Electrical Engineering
University of Southern California
Los Angeles, California
pedram@usc.edu

ABSTRACT

Deep neural networks are among the most influential architectures of deep learning algorithms, being deployed in many mobile intelligent applications. End-side services, such as intelligent personal assistants (IPAs), autonomous cars, and smart home services often employ either simple local models or complex remote models on the cloud. Mobile-only and cloud-only computations are currently the status-quo approaches. In this paper, we propose an efficient, adaptive, and practical engine, JointDNN, for collaborative computation between a mobile device and cloud for DNNs in both inference and training phase. JointDNN not only provides an energy and performance efficient method of querying DNNs for the mobile side, but also benefits the cloud server by reducing the amount of its workload and communications compared to the cloud-only approach. Given the DNN architecture, we investigate the efficiency of processing some layers on the mobile device and some layers on the cloud server. We provide optimization formulations at layer granularity for forward and backward propagation in DNNs, which can adapt to mobile battery limitations and cloud server load constraints and quality of service. JointDNN achieves up to 18× and 32× reductions on the latency and mobile energy consumption of querying DNNs compared to the status-quo approaches, respectively.

KEYWORDS

mobile cloud computing, deep learning, deep neural networks, intelligent services

1 INTRODUCTION

Deep Neural Network (DNN) architectures are promising solutions in achieving remarkable results in a wide range of machine learning applications, including, but not limited to computer vision, speech recognition, language modeling and autonomous cars.

Currently, there is a major growing trend in introducing more advanced DNN architectures and employing them in end-user applications. The considerable improvements in DNNs are usually achieved by increasing complexity which requires more computational resources for training and inference. Recent research directions to make this progress sustainable are: development of Graphical Processing Units (GPUs) as the vital hardware component of both servers and mobile devices [27], design of efficient algorithms for large-scale distributed training [7] and efficient inference [33], compression and approximation of models [38], and most recently introducing collaborative computation of cloud and fog as known as dew computing [36].

Using cloud servers for computation and storage is becoming extensively favorable due to technical advancements and improved accessibility. Scalability, low cost, and satisfactory Quality of Service (QoS), made offloading to cloud the typical choice for computing intensive tasks. On the other side, mobile-device are being equipped with more powerful general purpose CPUs and GPUs. Very recently there is a new trend in hardware companies to design dedicated chips to better tackle machine-learning tasks. For example, Apple's A11 Bionic chip [26] used in iPhone X uses a neural engine in its GPU to speed up DNN queries of applications such as face identification and facial motion capture [23].

There are currently two methods for DNN inference: mobile only and cloud only. In simple models, a mobile device is responsible for performing all of the computation. In case of complex models, the raw input data (image, video stream, voice, etc.) is uploaded and then computed on the cloud. The results of the task are later downloaded to the device.

Besides the improvements of the mobiles devices mentioned earlier, the computational power of mobile devices are still considered significantly weaker than the cloud ones. Therefore, mobile-only approach can cause large inference latency and failure in meeting QoS. Moreover, embedded devices undergo major energy consumption constraints due to battery capacity limits. On the other hand, cloud-only suffers communication overhead for uploading the raw data and downloading the outputs. Moreover, slowdowns caused by service congestions, subscription costs, and network dependency should be considered as downsides of this approach.

The superiority and persistent improvement of DNNs is heavily dependent on providing huge amount of training data. Typically, this data is collected from different resources and later fed into network for training. The final model can then be delivered to different devices for inference functions. However, there is a trend of appearance of applications requiring adaptive learning in online environments, such as self driving cars and security drones [30][25]. Model parameters in these smart devices are constantly being changed based on their continuous interaction with surroundings. Complexity of these architectures with extended number of parameters and current cloud-only methods for DNN training, implies a constant communication cost and burden of increased power consumption for mobile device.

Automatic partitioning of computationally extensive tasks over the cloud for optimization of performance and energy consumption has been already well studied [2]. Most recently, scalable distributed hierarchy structures between end-user device, edge, and cloud have been suggested [39] which are specialized for DNN applications.

However, exploiting the layer granularity of DNN architectures for run time partitioning has not been studied throughly yet.
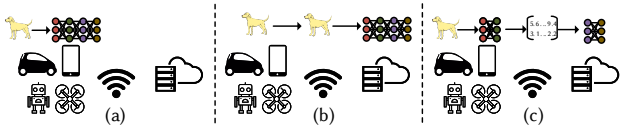

Figure 1: Different computation partitioning methods. (a) Mobile only: computation is completely done on mobile device. (b) Cloud only: raw input data is sent to cloud, computations is done on cloud and results are sent back to mobile device. (c) JointDNN: DNN architecture is partitioned at the granularity of layers, each layer can be computed either on cloud or mobile.

In this work, we are investigating inference and training of DNNs in a **joint** platform of mobile and cloud as an alternatives to the current single-platform methods as illustrated in Figure 1. Considering DNN architectures as an ordered sequence of layers, and possibility of computation of every layer either on mobile or cloud, we can model the DNN structure as a directed acyclic graph (DAG). The parameters of our real-time adaptive model are dependent on the following factors: mobile/cloud hardware and software resources, battery capacity, network specifications, and QoS. Based on this modeling, we show that the problem of finding the optimal computation schedule for different scenarios, i.e. best performance or energy consumption, can be reduced to the polynomial time shortest path problem.

To present realistic results, we made experiments with real hardwares as mobile device and cloud. To model the communication between platform, we used different network technologies and the most recent reports on their specifications in the U.S.

DNN architectures can be categorized based on functionality. These differences enforce specific type and order of layers in architecture, directly affecting the partitioning result in the collaborative method. For discriminative models, used in recognition applications, the layer size gradual decrease proceeding from input toward output 2. This sequence suggests computation of the first few layers on the mobile device to avoid excessive communication cost of uploading large raw input data. On the other hand, growth of the layer size from input to output in generative models used for synthesizing new data, implies the possibility of uploading small input to the cloud and later downloading and computing the last layers on the mobile device for better efficiency. Interesting mobile applications like image to image translation are implemented with autoencoder architectures, usually consisting of middle layers with smaller sizes compared to input and output. Consequently we expect the first and last layers to be computed on the mobile device in our collaborative approach. We examined eight well-known DNN benchmarks selected from these categories to illustrate their differences in collaborative computation approach.

As we will see in Section ??, the communication between the mobile and cloud is the main bottleneck for both performance and energy in the collaborative approach. We investigated the specific characteristics of CNN layer outputs and introduced a lossless compression method to reduce the communication costs.
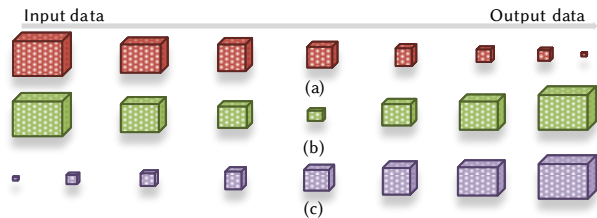

Figure 2: Typical layer size architecture of (a) Discriminative (b) Autoencoder (c) Generative models.

State-of-the-art work for collaborative computation of DNNs [21] only considers one offloading point, assigning computation of its previous layers and next layers on the mobile and cloud platforms, respectively. We show that this approach is non-generic and fails to be optimal, and introduced a new method granting the possibility of computation on either platforms for each layer independent of other layers. Our evaluations show that JointDNN significantly improves the latency and energy up to 3× and 7× respectively compared to the status-quo single platform approaches without any compression. The main contributions of this paper can be listed as:

- Introducing a novel model for collaborative computation between the mobile and cloud
- Formulating the problem of optimal computation scheduling of DNNs at layer granularity in mobile cloud computing environment as shortest path problem and integer linear programming (ILP)
- Examining compressibility of DNN layers and developing a lossless compression method to improve communication costs
- Demonstrating the significant improvements of performance, mobile energy consumption, and cloud workload achieved by using **JointDNN**

2 PROBLEM DEFINITION AND MODELING

In this section, we explain the general architecture of DNN layers and our profiling method. Moreover, we elaborate on how the cost optimization can be reduced to a shortest path problem by introducing the JointDNN graph model. Finally, we show how the constrained problem is formulated by setting up ILP.

2.1 DNN Building Blocks

DNNs are networks composed of several layers stacked to each other. We briefly explain the functionality of each layers used in the state-of-the-art architectures:

Convolution Layer (*conv*) consists of a set of filters with dimensions relatively smaller than their input. Each filter completely traverses through the input with a predefined step size and computes the dot product between it's parameters and the corresponding part of the input. This process creates different feature maps (referred to as channels) for different filters from the same input data. This aspect of preserving the locality of input features has made Convolutional Neural Network (CNN) architectures the horse power of the state-of-the-art image classification models. Because of dot product basis of *conv*, it can be formulated as General Matrix

Multiplication (GEMM), therefore capable of gaining performance improvement by using parallel computing devices (e.g. GPUs).

Fully Connected Layer *(fc)* is the main component of most regular neural networks in which every neuron is connected to all neurons of the previous layer. This fully pairwise connection architecture comprises large portion of computation of the whole network. Like *conv*, *fc* layer is also formulated as GEMM.

Pooling Layer *(pool)* performs a non-linear down sampling function over non-overlapping spatially local parts of input. Max-pooling is the most common function used in this type of layer alongside other functions such as average or L2-norm pooling.

Activation Layer increases the non-linearity property of neural network architectures. This layer applies non-linear activation function on single data points of input to generate an output with the same size. Among various non-linear functions, such as sigmoid and hyperbolic tangent, Rectified Linear Unit *(relu)* is currently the favorable choice in DNN architectures as it is simple and speeds up the tedious training process [10].

Local Response Normalization *(lrn)* performs local normalization by imposing a local competition for big activities between adjacent features in a channel, and also between features at the same spatial location in different channels. *lrn* are inspired by inhibition schemes observed in the brain helps with intention of generalization. There are different formulations suggested for *lrn*, as shown in [19, 22] they may lead to slight improvements.

Dropout Layer *(drop)* As mentioned earlier, *fc* occupies most of the parameters of DNN models and thus vulnerable to overfitting. Typically regularization methods are used to prevent overfitting by reducing high dependency of network on individual neurons during training. In dropout [37] technique, at each training iteration every neurons can be removed (droped out) from network with a predetermined probability p or kept with probability $1 - p$ and the training is done on the remaining network. The dropped out nodes will have their previous weight for the next training iteration.

Deconvolution Layer *(deconv)* also known as transposed convolution is mostly used on generative and autoencoder models in applications such as building high-resolutions picture from low-resolution pictures and high-level descriptions. The goal in deconvolution is to find f in the convolution equation of form $f * g = h$. In case of DNNs, g is the filter and f is the input of the convolution [42].

Long Short-Term Memory Layer *(lstm)* is a building unit for layers of a recurrent neural network (RNN) and is widely used due to its promising results in speech recognition applications. A typical LSTM unit is composed of a cell, an input gate, an output gate and a forget gate, which is responsible for remembering and forgetting specific values over arbitrary time intervals. The whole LSTM unit can be thought as a typical artificial neuron, as in a feed-forward neural network.

Softmax *(soft)* is the last layer in multi-class architectures, usually connected in a one-to-one correspondence way to a *fc* layer. Softmax establishes a probability distribution by representing each class probability with a single neuron.
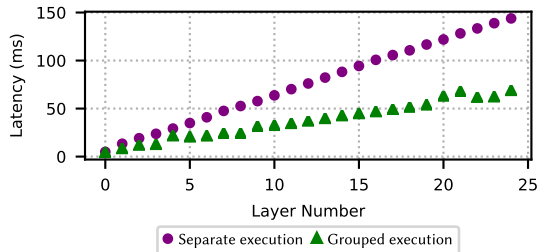

Figure 3: Latency of grouped and separated execution of convolution operator.

2.2 Energy and Latency Profiling

There are three methods in measuring the latency and energy consumption of each layer in neural networks:

Statistical Modeling: In this method, a regression model over the configurable parameters of operators (e.g. filter size in convolution) can be used to estimate the associated latency and energy. This method is prone to large error because of the inter-layer optimizations performed by DNN software packages. Therefore, it is necessary to consider execution of several consecutive operators grouped with each other during profiling. Many of these software packages are proprietary, making access to inter-layer optimization techniques impossible.

In order to illustrate this issue, we designed two experiments with 25 consecutive convolutions on NVIDIA Pascal™ GPU using cuDNN® library [1]. In the first experiment, we measure the latency of each convolution operator separately and set the total latency as sum of them. In the second experiment, we group the convolutions together and measure the total latency. All parameters are located on GPU's memory in both experiments, avoiding any data transfer from the main memory to make sure results are exactly representing the actual computation latency.

As we see in Figure 3, there is a large error gap between separated and grouped execution experiments which grows as the number of convolutions is increased. This observation confirms that we need to profile grouped operators to have more accurate estimations. Considering various consecutive combination of operators and different input sizes, this method requires a very large number of measurements, not to mention the need for a complex regression model.

Analytical Modeling: To derive an analytical approach for estimation of the latency and energy consumption, it is required to obtain the exact hardware and software specifications. However, the state-of-the-art work in latency modeling of DNNs [31] fails to estimate layer-level delay within an acceptable error bound, for instance, underestimating the latency of a fully connected layer with 4096 neurons by around 900%. Industrial developers do not reveal the detailed hardware architecture specifications and the proprietary parallel computing architectures such as CUDA®, therefore, analytical approach could be quite challenging [15].

Application-specific Profiling: In this method, the DNN architecture of the application being used is profiled in run-time. The number of applications in a mobile device using neural networks
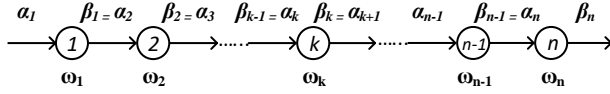
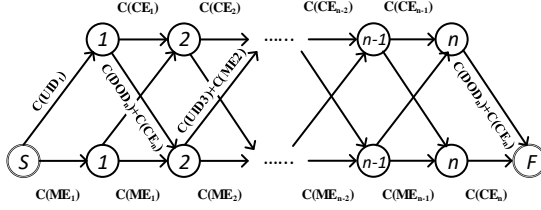**Figure 4: Computation model in linear topology.**

Figure 5: Graph representation of mobile cloud computing optimal scheduling problem for linear topology.

are generally limited. In conclusion, this method is more feasible, promising higher accuracy estimations. We have chosen this method for estimation of energies and latencies in the experiments of this paper.

2.3 JointDNN Graph Model

First, we assume that a DNN is presented by a sequence of distinct layers with a linear topology as depicted in Figure 4. Layers are executed sequentially, with output data generated by one layer feeds into the input of the next one. We denote the input and output data sizes of kth layer as α_k and β_k, respectively. Denoting the latency (energy) of layer k as ω_k, where $k = 1, 2, ..., n$, the total latency (energy) of querying the DNN is $\sum_{k=1}^{n} \omega_k$.

The mobile cloud computing optimal scheduling problem can be reduced to a shortest path problem, from node S to F, in the graph of Figure 5. **Mobile Execution** cost of the kth layer ($C(ME_k)$) is the cost of executing the kth layer in the mobile while the cloud server is idle. **Cloud Execution** cost of the kth layer ($C(CE_k)$) is the executing cost of the kth layer in the cloud server while the mobile is idle. **Uploading the Input Data** cost of the kth layer is the cost of uploading output data of the $(k\text{-}1)^{th}$ layer to the cloud server (UID_k). **Downloading the Input Data** cost of the kth layer is the cost of downloading output data of the $(k\text{-}1)^{th}$ layer to the mobile (DID_k). The costs can refer to either latency or energy. However, as we showed in Section 2.2, the assumption of linear topology in DNNs is not true and we need to consider all the consecutive grouping of the layers in the network. This fact suggests replacement of linear topology by a tournament graph as depicted in Figure 6. We define the parameters of this new graph, *JointDNN graph model*, in Table 1.

In this graph, node $C_{i:j}$ represents that the layers i to j are computed on the cloud server, while node $M_{i:j}$ represents that the layers i to j are computed on the mobile device. An edge between two adjacent nodes in JointDNN graph model is associated with four possible cases: 1) A transition from the mobile to the mobile, which only includes the mobile computation cost ($ME_{i,j}$) 2) A transition from the cloud to the cloud, which only includes the cloud computation cost ($CE_{i,j}$) 3) A transition from the mobile to the cloud, which

includes the mobile computation cost and uploading cost of the inputs of the next node ($EU_{i,j} = ME_{i,j} + UID_{j+1}$) 4) A transition from the cloud to the mobile, which includes the cloud computation cost and downloading cost of the inputs of the next node ($ED_{i,j} = CE_{i,j} + DID_{j+1}$). Under this formulation, we can transform the computation scheduling problem to finding the shortest path from S to F.

Residual networks are a class of powerful and easy-to-train architectures of DNNs [14].

In residual networks, as depicted in Figure 7 (a), the output of one layer is fed into another layer with distance of at least two. Thus, we need to keep track of the source layer (node 2 in Figure 7) so as to know that this layer is computed on the mobile or the cloud.

Our standard graph model has a memory of one which is the very previous layer. We provide a method to transform the computation graph of this type of network to our standard model, JointDNN graph.

In this regard, we add two additional chains of size $k - 1$, where k is the number of nodes in the residual block (3 in Figure 7). One chain represents the case of computing layer 2 on the mobile and the other one represents the case of computing layer 2 on the cloud. In Figure 7, we have only shown the weights that need to be modified, where D_2 and U_2 are the cost of downloading and uploading the output of layer 2, respectively.

By solving the shortest path problem in JointDNN graph model, we can obtain the optimal scheduling of inference in DNNs. Online training consists of one inference and one back-propagation step. The total number of layers is noted by N consistently throughout this paper so there are $2N$ layers for modeling training, where the second N layers are the mirrored version of the first N layers, and their associated operations are the gradients of the error function with respect to the DNN's weights. The main difference between the mobile cloud computing graph of inference and online training is the need for updating the model by downloading the new weights from the cloud. We assume that the cloud server performs the whole back-propagation step separately, even if it is scheduled to be done on the mobile, therefore, there is no need for mobile device to upload the weights that are updated by itself in order to save mobile energy consumption. The modification in JointDNN graph

Table 1: Parameter Definition of Graph Model

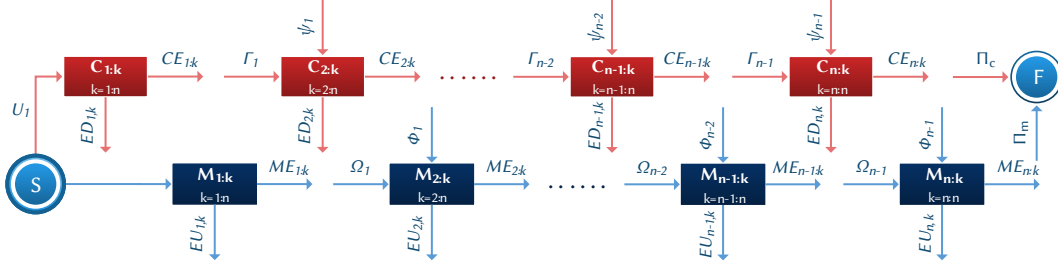| Param.	Description of Cost
$CE_{i:j}$	Executing layers i to j on the cloud
$ME_{i:j}$	Executing layers i to j on the mobile
$ED_{i,j}$	$CE_{i:j} + DID_j$
$EU_{i,j}$	$ME_{i:j} + UID_j$
ϕ_k	All the following edges: $\forall i = 1 : k - 1\ ED_{i,k-1}$
Ω_k	All the following edges: $\forall i = 1 : k - 1\ ME_{i,k-1}$
Ψ_k	All the following edges: $\forall i = 1 : k - 1\ EU_{i,k-1}$
Γ_k	All the following edges: $\forall i = 1 : k - 1\ CE_{i,k-1}$
Π_m	All the following edges: $\forall i = 1 : n\ ME_{i,n}$
Π_c	All the following edges: $\forall i = 1 : n\ ED_{i,n}$
U_1	Uploading the input of the first layer

Figure 6: JointDNN graph model.

model is adding the costs of downloading weights of the layers that are updated in the cloud to $ED_{i,j}$.

The shortest path problem can be solved in polynomial time efficiently.

However, the problem of shortest path subjected to constraints has been shown to be NP-Complete [41]. For instance, assuming our standard graph is constructed for energy and we need to find the shortest path subject to the constraint of the total latency of that path being less than a time deadline (QoS). However, there is an approximation solution to this problem, "LARAC" algorithm [20], the nature of our application does not require to solve this optimization problem frequently, therefore, we aim to obtain the optimal solution. We can constitute a small look-up table of optimization results for different set of parameters (e.g. network bandwidth, cloud server load, etc.). We provide the ILP formulations of DNN partitioning in the following sections.

2.4 ILP Setup

2.4.1 Performance Efficient Computation Offloading ILP Setup for Inference. We formulated the scheduling of inference in DNNs as an ILP with tractable number of variables. In our method, first we profile the delay and energy consumption of consecutive layers

Figure 7: (a) A residual building block (b) Transformation of a residual building block into shortest path problem.

of size $m \in \{1, 2, \ldots, N\}$. Thus, we will have

$$N + (N - 1) + \ldots + 1 = N(N + 1)/2 \quad (1)$$

number of different profiling values for delay and energy. Considering layer i to layer j to be computed either on the mobile device or cloud server, we assign two binary variables $m_{i,j}$ and $c_{i,j}$, respectively. Download and upload communication delays needs to be added to the execution time, when switching from/to cloud to/from mobile, respectively.

$$T_{computation} = \sum_{i=1}^{n} \sum_{j=i}^{n} (m_{i,j}.T_{mobile_{L_{i,j}}} + c_{i,j}.T_{cloud_{L_{i,j}}}) \quad (2)$$

$$
\begin{aligned}
T_{communication} = & \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=j+1}^{n} m_{i,j}.c_{j+1,k}.T_{upload_{L_j}} \\
& + \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=j+1}^{n} c_{i,j}.m_{j+1,k}.T_{download_{L_j}} \\
& + \sum_{i=1}^{n} c_{1,i}.T_{upload_{L_i}} \\
& + \sum_{i=1}^{n} c_{i,n}.T_{download_{L_n}}
\end{aligned} \quad (3)
$$

$$T_{total} = T_{computation} + T_{communication} \quad (4)$$

$T_{mobile_{L_{i,j}}}$ and $T_{cloud_{L_{i,j}}}$ represent the execution time of the ith layer to the jth layer on the mobile and cloud, respectively. $T_{download_{L_i}}$ and $T_{upload_{L_i}}$ represent the latency of downloading and uploading the output of the ith layer, respectively. Considering each set of the consecutive layers, whenever $m_{i,j}$ and one of $\{c_{j+1,k}\}_{k=j+1:n}$ are equal to one, the output of the jth layer is uploaded to the cloud. The same argument applies to downloading. We also note that the last two terms in Eq. 3 represent the condition by which the last layer is computed on the cloud and we need to download the output to the mobile device, and the first layer is computed on the cloud and we need to upload the input to the cloud, respectively. To support for residual architectures, we need to add a pair of download and upload terms similar to the first two terms in Eq. 3 for the starting and ending layers of each residual block. In order to guarantee that all layers are computed exactly once, we need to add the following set of constraints:
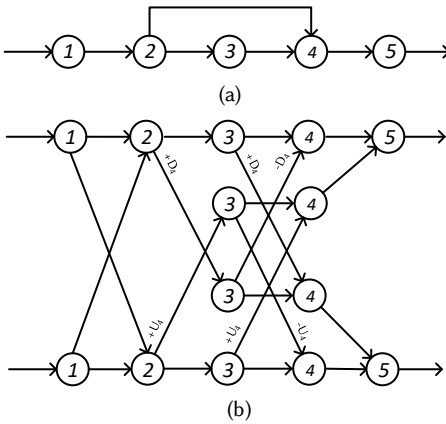
$$\forall m \in 1:n : \sum_{i=1}^{m} \sum_{j=m}^{n} (m_{i,j} + c_{i,j}) = 1 \qquad (5)$$

Because of the non-linearity of multiplication, an additional step is needed to transform Eq. 3 to the standard form of ILP. We define two sets of new variables:

$$u_{i,j} = m_{i,j}. \sum_{k=j+1}^{n} c_{j+1,k}$$
$$d_{i,j} = c_{i,j}. \sum_{k=j+1}^{n} m_{j+1,k} \qquad (6)$$

with the following constraints:

$$u_{i,j} \leq m_{i,j}$$
$$u_{i,j} \leq \sum_{k=j+1}^{n} c_{j+1,k}$$
$$m_{i,j} + \sum_{k=j+1}^{n} c_{j+1,k} - u_{i,j} \leq 1$$
$$d_{i,j} \leq c_{i,j} \qquad (7)$$
$$d_{i,j} \leq \sum_{k=j+1}^{n} m_{j+1,k}$$
$$c_{i,j} + \sum_{k=j+1}^{n} m_{j+1,k} - d_{i,j} \leq 1$$

The first two constraints ensure that $u_{i,j}$ will be zero if either $m_{i,j}$ or $\sum_{l=j+1}^{n} c_{j+1,l}$ are zero. The third inequality guarantees that $u_{i,j}$ will take value one if both binary variables, $m_{i,j}$ and $\sum_{l=j+1}^{n} c_{j+1,l}$, are set to one. The same reasoning works for $d_{i,j}$. In summary, the total number of variables in our ILP formulation will be $4N(N+1)/2$, where N is total number of layers in the network.

2.4.2 Energy Efficient Computation Offloading ILP Setup for Inference. Because of the nature of the application, we only care about the energy consumption on the mobile side. We formulate ILP as follows:

$$E_{computation} = \sum_{i=1}^{n} \sum_{j=i}^{n} m_{i,j}.E_{mobile_{L_{i,j}}} \qquad (8)$$

$$E_{communication} = \sum_{i=2}^{n} \sum_{j=i}^{n} m_{i,j}.E_{download_{L_i}}$$
$$+ \sum_{i=1}^{n} \sum_{j=i}^{n-1} m_{i,j}.E_{upload_{L_j}} \qquad (9)$$
$$+ (\sum_{i=1}^{n} (1 - m_{1,i}) - (n-1)).E_{upload_{L_1}}$$
$$+ (\sum_{i=1}^{n} (1 - m_{i,n}) - (n-1)).E_{download_{L_n}}$$

$$E_{total} = E_{computation} + E_{communication} \qquad (10)$$

$E_{mobile_{L_{i,j}}}$ and $E_{cloud_{L_{i,j}}}$ represent the amount of energy required to compute the ith layer to the jth layer on the mobile and cloud, respectively. $E_{download_{L_i}}$ and $E_{upload_{L_i}}$ represent the energy required to download and upload the output of ith layer, respectively. Similar to performance efficient ILP constraints, each layer should be executed exactly once:

$$\forall m \in 1:n : \sum_{i=1}^{m} \sum_{j=m}^{n} m_{i,j} \leq 1 \qquad (11)$$

The ILP problem can be solved for different set of parameters (e.g. different uplink and download speeds), and then the scheduling results can be stored as a look-up table in the mobile device. Moreover because the number of variables in this setup is tractable solving ILP is quick. For instance, solving ILP for AlexNet takes around 0.045 seconds on Intel(R) Core(TM) i7-3770 CPU with MATLAB®'s intlinprog() function using primal simplex algorithm.

2.4.3 Performance Efficient Computation Offloading ILP Setup for Training. The ILP formulation of online training phase is very similar to that of inference. In online training we have $2N$ layers instead of N obtained by mirroring the DNN, where the second N layers are backward propagation. Moreover, we need to download the weights that are updated in the cloud to the mobile. We assume that the cloud server always has the most updated version of the weights and does not require the mobile device to upload the updated weights. The following terms need to be added for the ILP setup of training:

$$T_{computation} = \sum_{i=1}^{2n} \sum_{j=i}^{2n} (m_{i,j}.T_{mobile_{L_{i,j}}} + c_{i,j}.T_{cloud_{L_{i,j}}}) \qquad (12)$$

$$T_{communication} = \sum_{i=1}^{2n} \sum_{j=i}^{2n} \sum_{k=j+1}^{2n} m_{i,j}.c_{j+1,k}.T_{upload_{L_j}}$$
$$+ \sum_{i=1}^{2n} \sum_{j=i}^{2n} \sum_{k=j+1}^{2n} c_{i,j}.m_{j+1,k}.T_{download_{L_j}}$$
$$+ \sum_{i=1}^{n} c_{1,i}.T_{upload_{L_i}} \qquad (13)$$
$$+ \sum_{i=n+1}^{2n} \sum_{j=i}^{2n} c_{i,j}.T_{download_{W_i}}$$

$$T_{total} = T_{computation} + T_{communication} \qquad (14)$$

2.4.4 Energy Efficient Computation Offloading ILP Setup for Training.

$$E_{computation} = \sum_{i=1}^{2n} \sum_{j=i}^{2n} m_{i,j}.E_{mobile_{L_{i,j}}} \qquad (15)$$

$$E_{communication} = \sum_{i=2}^{2n} \sum_{j=i}^{2n} m_{i,j}.E_{download_{L_i}}$$

$$+ \sum_{i=1}^{2n} \sum_{j=i}^{2n-1} m_{i,j}.E_{upload_{L_j}}$$

$$+ (\sum_{i=1}^{2n} (1 - m_{1,i}) - (2n-1)).E_{upload_{L_1}}$$

$$+ (\sum_{i=n+1}^{2n} \sum_{j=i}^{2n} (1 - m_{i,j}) - (n-1)).E_{download_{W_i}}$$

$$\tag{16}$$

$$E_{total} = E_{computation} + E_{communication} \tag{17}$$

2.4.5 Scenarios. There can be different optimization scenarios defined for ILP as listed below:

- **Performance efficient computation:** In this case, it is sufficient to solve the ILP formulation for performance efficient computation offloading.
- **Energy efficient computation:** In this case, it is sufficient to solve the ILP formulation for energy efficient computation offloading.
- **Battery budget limitation:** In this case, based on the available battery, the operating system can decide to dedicate a specific amount of energy consumption to each application. By adding the following constraint to the performance efficient ILP formulation, our framework would adapt to battery limitations:

$$E_{computation} + E_{communication} \leq E_{ubound} \tag{18}$$

- **Cloud limited resources:** In the presence of cloud server congestion or limitations on user's subscription, we can apply execution time constraints to each application to alleviate the server load:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} c_{i,j}.T_{cloud_{L_{i,j}}} \leq T_{ubound} \tag{19}$$

- **QoS:** In this scenario, we minimize the required energy consumption while meeting a specified deadline:

$$min\{E_{computation} + E_{communication}\}$$
$$T_{computation} + T_{communication} \leq T_{QoS} \tag{20}$$

This constraint could be applied to both energy and performance efficient ILP formulations.

3 EVALUATION

3.1 Deep Architecture Benchmarks

Since the architecture of neural networks depends on the type of the application, we have chosen three common application types of DNNs:

(1) **Discriminative neural networks** are a class of models in machine learning for modeling the conditional probability distribution $P(y|x)$. This class generally is used in classification and regression tasks. AlexNet[22], OverFeat[34],

---

Algorithm 1: JointDNN engine optimal scheduling of DNNs

1 function JointDNN (N, L_i, D_i, NB, NP);
    Input : 1: N: number of layers in the DNN
             2: $L_i|i = 1 : N$: layers in the DNN
             3: $D_i|i = 1 : N$: data size at each layer
             4: NB: mobile network bandwidth
             5: NP: mobile network uplink and downlink power consumption
    Output : Optimal schedule of DNN
2 **for** $i = 0$; $i < N$; $i = i + 1$ **do**
3 | **for** $j = 0$; $j < N$; $j = j + 1$ **do**
4 | | $Latency_{i,j}, Energy_{i,j}$ = ProfileGroupedLayers(i,j);
5 | **end**
6 **end**
7 G,S,F = ConstructShortestPathGraph(N,L_i,D_i,NB,NP) //S and F are start and finish nodes and G is the JointDNN graph model
8 **if** *no constraints* **then**
9 | *schedule* = **ShortestPath(G,S,F)**
10 **else**
11 | **if** *Battery Limited Constraint* **then**
12 | | $E_{comm} + E_{comp} \leq E_{ubound}$
13 | | *schedule* = PerformanceEfficientILP(N,L_i,D_i,NB,NP)
14 | **end**
15 | **if** *Cloud Server Contraint* **then**
16 | | $\sum_{i=1}^{n} \sum_{j=i}^{n} c_{i,j}.T_{cloud_{L_{i,j}}} \leq T_{ubound}$
17 | | *schedule* = PerformanceEfficientILP(N,L_i,D_i,NB,NP)
18 | **end**
19 | **if** *QoS* **then**
20 | | $T_{comm} + T_{comp} \leq T_{QoS}$
21 | | *schedule* = EnergyEfficientILP(N,L_i,D_i,NB,NP)
22 | **end**
23 | ;
24 **end**
25 **return** *schedule*;

VGG16[35], Deep Speech[13], ResNet[14], and NiN[24] are well-known discriminative models we use as benchmarks in this experiment. Except Deep Speech, used for speech recognition, all other benchmarks are used in image classification tasks.

(2) **Generative neural networks** model the joint probability distribution $P(x, y)$, allowing generation of new samples. These networks have applications in Computer Vision [11] and Robotics [9], which can be deployed on a mobile device. Chair [8] is a generative model we use as benchmark in this work.

(3) **Autoencoders** are another class of neural networks used to learn a representation for a data set. Their applications are image reconstruction, image to image translation, and denoising to name a few. Mobile robots can be equipped with autoencoders to be used in their computer vision tasks. We use Pix2Pix [18], as a benchmark from this class.
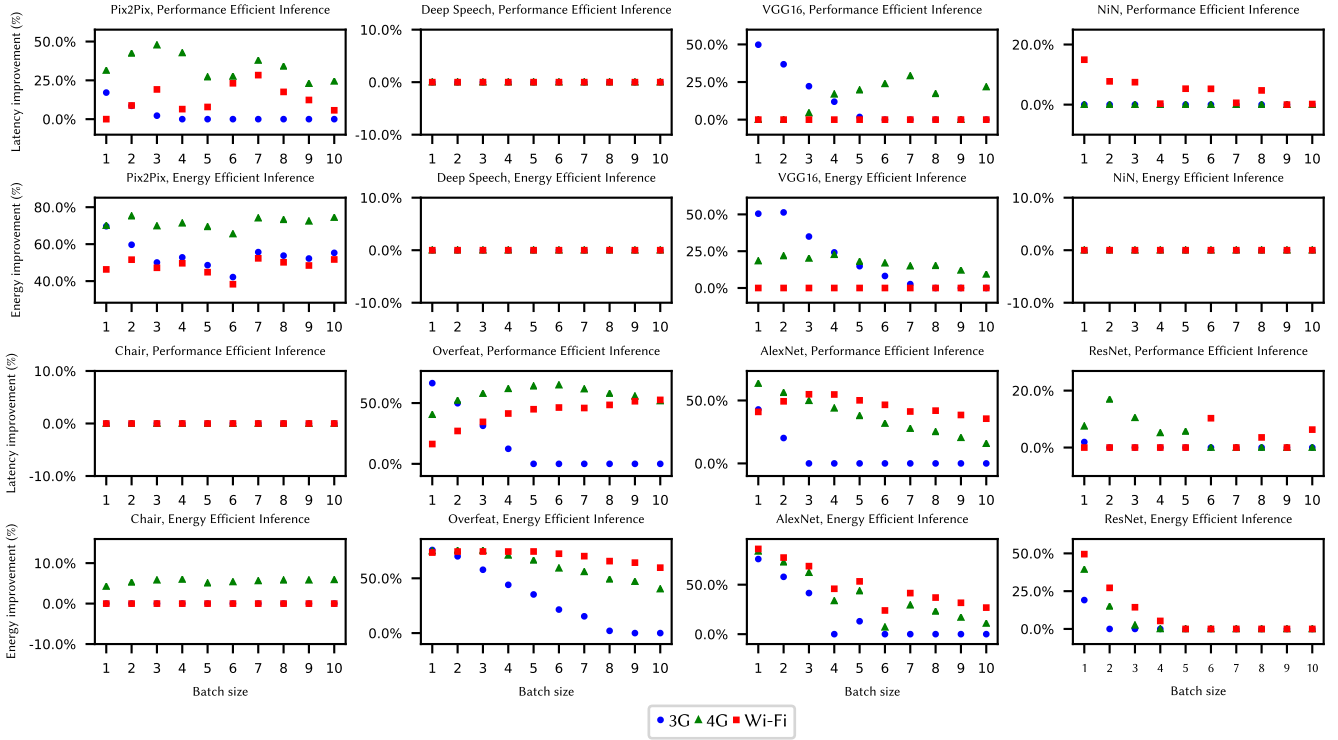
Figure 8: Latency and energy improvements for different batch sizes during inference.

Table 2: Benchmark Specifications

Type	Model	Layers
Discriminative	AlexNet	21
	OverFeat	14
	Deep Speech	10
	ResNet	70
	VGG16	37
	NiN	29
Generative	Chair	10
Autoencoder	Pix2Pix	32

Table 3: Mobile networks specifications in the U.S.

Param.	3G	4G	Wi-Fi
Download speed (Mpbs)	2.0275	13.76	54.97
Upload speed (Mbps)	1.1	5.85	18.88
α_u (mW/Mpbs)	868.98	438.39	283.17
α_d (mW/Mpbs)	122.12	51.97	137.01
β (mW)	817.88	1288.04	132.86

3.2 Mobile and Server Setup

We used Jetson TX2 module developed by NVIDIA® [3], a fair representative of mobile computation power as our mobile device. This module enables efficient implementation of DNN applications used in products such as robots, drones, and smart cameras. It is equipped with NVIDIA Pascal®GPU with 256 CUDA cores and a shared 8 GB 128 bit LPDDR4 memory between GPU and CPU. To measure the power consumption of the mobile platform, we used INA226 power sensor [17].

NVIDIA® Tesla® K40C [4] with 12 GB memory serves as our server GPU. The computation capability of this device is more than one order of magnitude compared to our mobile device.

3.3 Communication Parameters

To model the communication between platforms, we used the average download and upload speed of mobile Internet [28, 29] for different networks (3G, 4G and Wi-Fi) as shown in Table 3.

The communication power for download (P_d) and upload (P_u) is dependent on the network throughput (t_d and t_u). Comprehensive examinations in [16] indicates that uplink and downlink power can be modeled with linear equations (Eq. 21) fairly accurate with less than 6% error rate. Table 3 shows the parameter values of this equation for different networks.

$$P_u = \alpha_u t_u + \beta$$
$$P_d = \alpha_d t_d + \beta$$
(21)

4 RESULTS

The latency and energy improvements of inference and online training with our engine for 8 different benchmarks are shown in
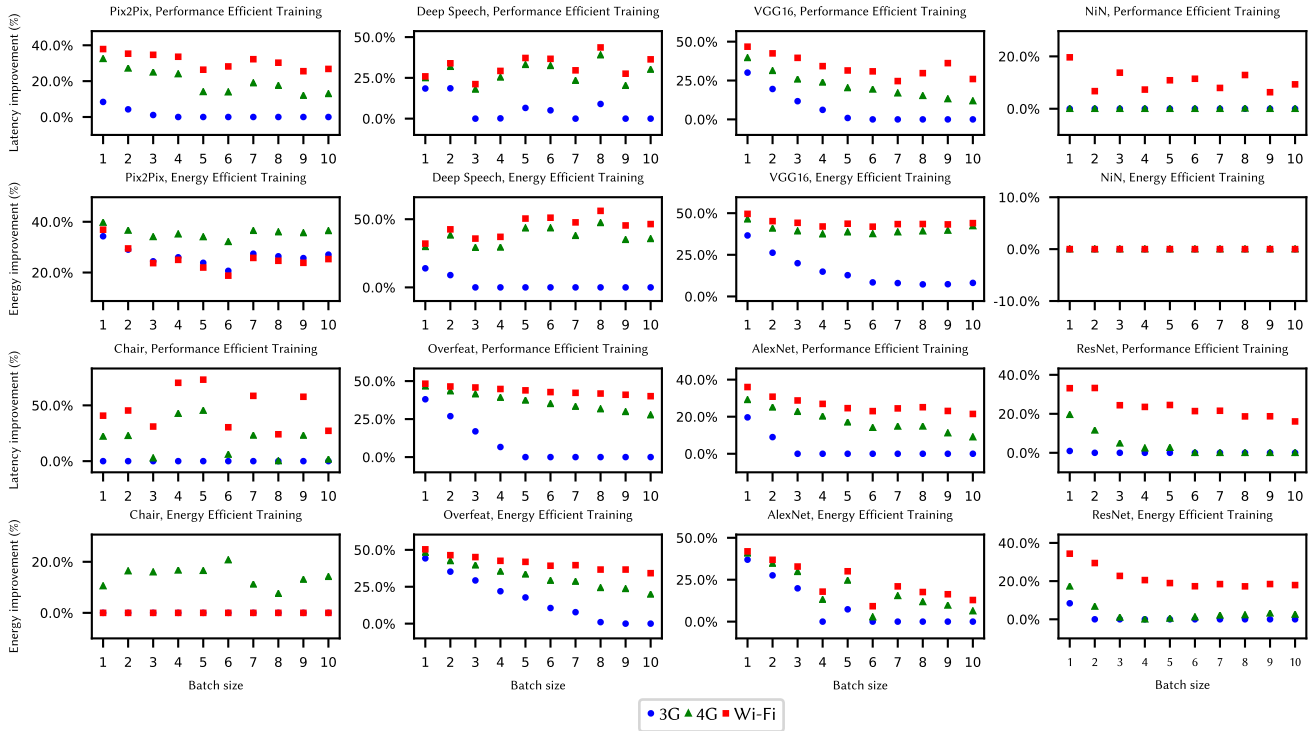
Figure 9: Latency and energy improvements for different batch sizes during training.

Figures 8 and 9, respectively. We considered the best case of mobile-only and cloud-only as our baseline. JointDNN can achieve up to 66% and 86% improvements in latency and energy consumption, respectively during inference. Communication cost increases linearly with batch size while this is not the case for computation cost and it grows with much lower rate, as depicted in 10(b). Therefore, a key observation is that as we increase the batch size, the mobile-only approach becomes more preferable.

During online training, the huge communication overhead of transmitting the updated weights will be added to the total cost. Therefore, in order to avoid downloading this large data, only a few back-propagation steps are computed in the cloud server. We performed a simulation by varying the percentage of updated weight. As the percentage of updated weights increases, the latency and energy consumption becomes constant which is shown in Figure 10. This is the result of the fact that all the back-propagations will be performed on the mobile device and weights are not transfered from the cloud to the mobile. JointDNN can achieve improvements up to 73% in latency and 56% in energy consumption during inference.

Different patterns of scheduling are demonstrated in Figure 11. They represent the optimal solution in Wi-Fi network while optimizing for latency. They show how the computations in DNN is divided between the mobile and the cloud. As it can be seen, discriminative models (e.g. AlexNet), inference follows a mobile-cloud pattern and training follows a mobile-cloud-mobile pattern. The intuition is that the last layers are computationally intensive *(fc)* with small data sizes, which require a low communication cost,

therefore, last layers tend to be computed on the cloud. For generative models (e.g. Chair), the execution schedule of inference is
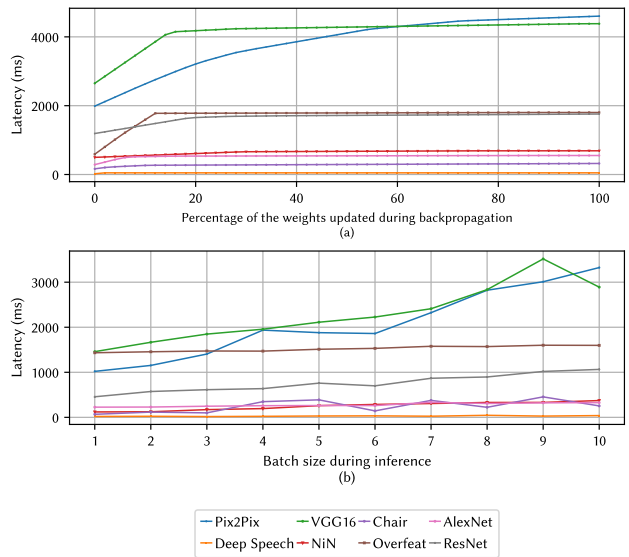

Figure 10: (a) Latency of one epoch of online training using JointDNN algorithm vs percentage of updated weights (b) Latency of mobile-only inference vs. batch size.
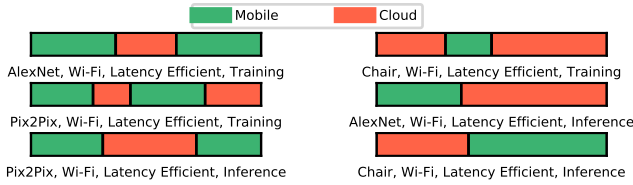
Figure 11: Interesting schedules of execution for three types of DNN architectures.
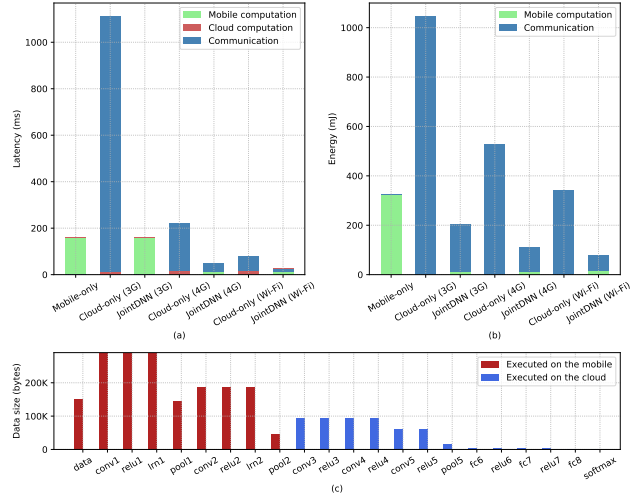

Figure 12: (a) Execution time of AlexNet optimized for performance (b) Mobile energy consumption of AlexNet optimized for energy (c) Data size of the layers in AlexNet and the scheduled computation, where the first nine layers are computed on the mobile and the rest on the cloud, which is the optimal solution w.r.t. both performance and energy.

the opposite of discriminative networks, in which the last layers are generally huge and in the optimal solution they are computed on the mobile. Lastly, for autoencoders, where both the input and output data sizes are large, the first and last layers are computed on the mobile.

JointDNN pushes some parts of the computations toward the mobile device. As a result this will lead to less workload on the cloud server. As we see in Table 4, we can reduce the cloud server's workload up to 84% and 53% on average, which enables the cloud provider to service more users, while obtaining higher performance and lower energy consumptions compared to single-platform approaches.

Table 4: Workload reduction of the cloud server in different mobile networks

Optimization Target	3G (%)	4G (%)	Wi-Fi (%)
Latency	84	49	12
Energy	73	49	51

4.1 Communication Dominance

Execution time and energy breakdown for AlexNet, which is noted as a representative for the state-of-the-art architectures deployed in cloud servers, is depicted in Figure 12. The cloud-only approach is dominated by the communication costs. As demonstrated in Figure 12, 99%, 93% and 81% of the total execution time is used for communication in case of 3G, 4G, and Wi-Fi, respectively. This relative portion also applies to energy consumption. Comparing the latency and energy of the communication to those of mobile-only approach, we notice that mobile-only approach for AlexNet is better than the cloud-only approach in all the mobile networks. We apply lossless compression methods in order to reduce the effect of the communication, which will be covered in the next section.

4.2 Layer Compression

The preliminary results of our experiments show that more than 75% of the total energy and delay cost in DNNs are caused by communication in the collaborative approach. This cost is directly proportional to the size of the layer being downloaded to or uploaded from the mobile device. Because of the complex feature extraction process of DNNs, the size of some of the intermediate layers are even larger than network's input data. For example, this ratio can go as high as 10× in VGG16. To address this bottleneck,

we investigated compression of the data before any communication. This process can be applied to different DNN architecture types; however, we only considered CNNs due to their specific characteristics explained later in details.

CNN architectures are mostly used for image and video recognition applications. Because of the spatially local preservation characteristics of *conv* layers, we can assume that the output of the first convolution layers are following the same structure as the
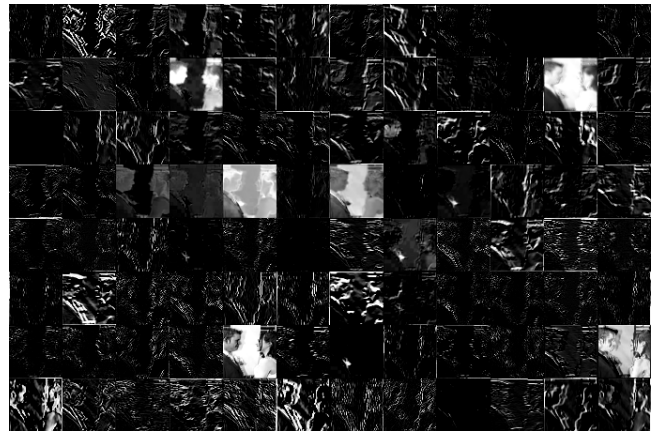

Figure 13: Layer output after passing the input image through *conv, relu* and *lrn*. Channels are preserving the general structure of the input image and large ratio of the output data is black (zero) due to existence of *relu*. Tiling is used to put all 96 channels together.

input image, as shown in Figure 13. Moreover, a big ratio of layer outputs are expected to be zero due to the presence of the relu layer. Our observations shows that the ratio of neurons equal to zero *(ZR)* varies from 50% to 90% after *relu* in CNNs. These two characteristics, layers being similar to the input image, and large proportion of their data being a single value, suggest that we can employ existing image compression techniques to their output.

There are two general categories of compression techniques, lossy and lossless [5]. In lossless techniques it is possible to reconstruct the original information completely. On the contrary, lossless techniques use approximations and the original data cannot be reconstructed. In our experiments, we examined the impact of compression using PNG, a lossless technique, based on encoding of frequent sequences in an image.

Even though the data type of DNN parameters in typical implementations are 32-bits floating-points, most image formats are based on 3-bytes RGB color triples. Therefore, to compress the layer in the same way as 2D pictures, the floating-point data should be quantized into 8-bits fixed-point. Recent studies show representing the parameters of DNNs with only 4-bits affect the accuracy not more than 1% [38]. In this work, we implemented our architectures with 8-bits fixed-point and presented our baseline without any compression. The layers of CNN contain numerous channels of 2D matrices, each similar to an image. A simple method is to compress each channel separately. In addition to extra overhead of file header for each channel, this method will not take the best of the frequent sequence decoding of PNG. One alternative is locating different channels side by side, referred to as tiling, to form a large 2D matrix representing one layer as shown in Figure 13. It should be noted that 1D *fc* layers are very small and we did not apply compression on them.

The Compression Ratio *(CR)* is defined as the ratio of the size of the layer (8-bit) to the size of the compressed 2D matrix in PNG. Looking at the results of compression for two different CNN architectures in Figure 14, we can observe a high correlation between ratio of pixels being zero *(ZR)* and *CR*. PNG can compress the layer data up to 5.8× and 3.5× by average. These results confirm the effectiveness of the proposed compression method. By replacing the compressed layers output and adding the cost of compression process itself in JointDNN formulations, we achieve an extra 4.9× and 4.6× improvements in energy and latency on average, respectively.

5 RELATED WORK AND COMPARISON

General Task Offloading Frameworks. There are existing prior arts focusing on offloading computation from the mobile to the cloud[2, 6, 12, 32, 40, 43]. However, all these frameworks share a limiting feature that makes them impractical for computation partitioning of the DNN applications.

These frameworks are programmer annotations dependent as they make decisions about pre-specified functions, whereas JointDNN makes scheduling decisions based on the model topology and mobile network specifications in run-time. Offloading in function level, cannot lead to efficient partition decisions due to layers of a given type within one architecture can have significantly different computation and data characteristics. For instance, a specific convolution
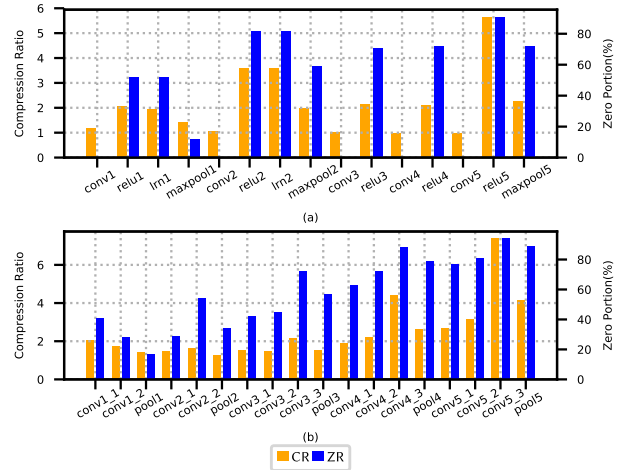

Figure 14: Compression Ratio (CR) and ratio of zero valued neurons (ZR) for different layers of (a) AlexNet and (b) VGG16.

layer structure can be computed on mobile or cloud in different models in the optimal solution.

Neurosurgeon is the only prior art exploring a similar computation offloading idea in DNNs between the mobile device and the cloud server at layer granularity. Neurosurgeon assumes that there is only one data transfer point and the execution schedule of the efficient solution starts with mobile and then switches to the cloud, which performs the whole rest of the computations. Our results show this is not true especially for online training, where the optimal schedule of execution often follows the mobile-cloud-mobile pattern. Moreover, generative and autoencoder models follow a multi data transfer points pattern. Also, the execution schedule can start with the cloud especially in case of generative models where the input data size is large. Furthermore, inter-layer optimizations performed by DNN libraries are not considered in Neurosurgeon. Moreover, Neurosurgeon only schedules for optimal latency and energy, while JointDNN adapts to different scenarios including battery limitation, cloud server congestion, and QoS. Lastly, Neurosurgeon only targets simple CNN and ANN models, while JointDNN utilizes a graph based approach to handle more complex DNN architectures like ResNet and RNNs.

6 CONCLUSIONS

In this paper, we demonstrated that the status-quo approaches, cloud-only or mobile-only, are not optimal with regard to latency and energy. We reduced the problem of partitioning the computations in a DNN to shortest path problem in a graph. Adding constraints to the shortest path problem makes it NP-Complete, therefore, we also provided ILP formulations to cover different possible scenarios of limitations of mobile battery, cloud congestion, and QoS. One can solve this problem for different set of parameters beforehand (e.g. network bandwidth, cloud server load, etc.) and use a look-up table accordingly to avoid the overhead of solving

the optimization problem. The output data size in discriminative networks is typically smaller than other layers in the network, therefore, last layers are expected to be computed on the cloud, while first layers are expected to be computed on the mobile. A reverse reasoning works for Generative models. Autoencoders have large input and output data sizes, which implies that the first and last layers are expected to be computed on the mobile. With these insights, the execution schedule of DNNs can possibly have various patterns depending on the model architecture.

ACKNOWLEDGMENTS

REFERENCES

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, and others. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[2] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. (2011), 301–314.

[3] Nvidia Corporation. 2018. Jetson TX2 Module. https://developer.nvidia.com/embedded/buy/jetson-tx2. (2018). [Online; accessed 15-January-2018].

[4] Nvidia Corporation. 2018. TESLA DATA CENTER GPUS FOR SERVERS. http://www.nvidia.com/object/tesla-servers.html. (2018). [Online; accessed 15-January-2018].

[5] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience.

[6] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, and others. 2010. MAUI: Making Smartphones Last Longer with Code Offload. (2010), 49–62. https://doi.org/10.1145/1814433.1814441

[7] Jeffrey Dean, Greg S. Corrado, Rajat Monga, and others. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1223–1231. http://dl.acm.org/citation.cfm?id=2999134.2999271

[8] Alexey Dosovitskiy, Jost Tobias Springenberg, and Thomas Brox. 2014. Learning to Generate Chairs with Convolutional Neural Networks. *CoRR* abs/1411.5928 (2014). arXiv:1411.5928 http://arxiv.org/abs/1411.5928

[9] Chelsea Finn and Sergey Levine. 2016. Deep Visual Foresight for Planning Robot Motion. *CoRR* abs/1610.00696 (2016). arXiv:1610.00696

[10] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Vol. 15. PMLR, Fort Lauderdale, FL, USA, 315–323. http://proceedings.mlr.press/v15/glorot11a.html

[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, and others. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2672–2680. http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[12] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. (2012), 93–106. http://dl.acm.org/citation.cfm?id=2387880.2387890

[13] Awni Y. Hannun, Carl Case, Jared Casper, and others. 2014. Deep Speech: Scaling up end-to-end speech recognition. *CoRR* abs/1412.5567 (2014). arXiv:1412.5567 http://arxiv.org/abs/1412.5567

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[15] Sunpyo Hong and Hyesoon Kim. 2010. An Integrated GPU Power and Performance Model. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 280–289. https://doi.org/10.1145/1816038.1815998

[16] Junxian Huang, Feng Qian, Alexandre Gerber, and others. 2012. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. ACM, New York, NY, USA, 225–238.

[17] Texas Instruments Incorporated. 2018. INA Current/Power Monitor. http://www.ti.com/product/INA226. (2018). [Online; accessed 15-January-2018].

[18] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. 2016. Image-to-Image Translation with Conditional Adversarial Networks. *CoRR* abs/1611.07004 (2016). arXiv:1611.07004 http://arxiv.org/abs/1611.07004

[19] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. 2009. What is the best multi-stage architecture for object recognition?. In *2009 IEEE 12th International Conference on Computer Vision*. 2146–2153.

[20] A. Juttner, B. Szviatovski, I. Mecs, and Z. Rajko. 2001. Lagrange relaxation based method for the QoS routing problem. 2 (2001), 859–868 vol.2.

[21] Yiping Kang, Johann Hauswald, Cao Gao, and others. 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 (ASPLOS '17)*. ACM, New York, NY, USA, 615–629. https://doi.org/10.1145/3037697.3037698

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[23] Hao Li, Jihun Yu, Yuting Ye, and Chris Bregler. 2013. Realtime Facial Animation with On-the-fly Correctives. *ACM Trans. Graph.* 32, 4, Article 42 (July 2013), 10 pages.

[24] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network In Network. *CoRR* abs/1312.4400 (2013). arXiv:1312.4400 http://arxiv.org/abs/1312.4400

[25] Mahdi Nazemi, Amir Erfan Eshratifar, and Massoud Pedram. 2018. A Hardware-Friendly Algorithm for Scalable Training and Deployment of Dimensionality Reduction Models on FPGA. In *Proceedings of the 19th IEEE International Symposium on Quality Electronic Design*.

[26] Apple Newsroom. 2017. The future is here: iPhone X. https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/. (2017). [Online; accessed 15-January-2018].

[27] Kyoung-Su Oh and Keechul Jung. 2004. GPU implementation of neural networks. 37 (06 2004), 1311–1314.

[28] OpenSignal.com. 2017. State of Mobile Networks: USA. https://opensignal.com/reports/2017/08/usa/state-of-the-mobile-network. (2017). [Online; accessed 15-January-2018].

[29] OpenSignal.com. 2017. United States Speedtest Market Report. http://www.speedtest.net/reports/united-states/. (2017). [Online; accessed 15-January-2018].

[30] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, and others. 2017. Agile Off-Road Autonomous Driving Using End-to-End Deep Imitation Learning. *CoRR* abs/1709.07174 (2017). arXiv:1709.07174 http://arxiv.org/abs/1709.07174

[31] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. (2017).

[32] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, and others. 2011. Odessa: Enabling Interactive Perception Applications on Mobile Devices. (2011), 43–56. https://doi.org/10.1145/1999995.2000000

[33] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing. 2017. LookNN: Neural network with no multiplication. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1775–1780. https://doi.org/10.23919/DATE.2017.7927280

[34] Pierre Sermanet, David Eigen, Xiang Zhang, and others. 2013. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *CoRR* abs/1312.6229 (2013). arXiv:1312.6229 http://arxiv.org/abs/1312.6229

[35] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). arXiv:1409.1556 http://arxiv.org/abs/1409.1556

[36] Karolj Skala, Davor Davidovic, Enis Afgan, Ivan Sovic, and Zorislav Sojat. 2015. Scalable Distributed Computing Hierarchy: Cloud, Fog and Dew Computing. *Open Journal of Cloud Computing (OJCC)* 2, 1 (2015), 16–24. http://nbn-resolving.de/urn:nbn:de:101:1-201705194519

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 1929–1958. http://dl.acm.org/citation.cfm?id=2627435.2670313

[38] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *CoRR* abs/1703.09039 (2017). arXiv:1703.09039 http://arxiv.org/abs/1703.09039

[39] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. Distributed Deep Neural Networks over the Cloud, the Edge and End Devices. *CoRR* abs/1709.01921 (2017). arXiv:1709.01921 http://arxiv.org/abs/1709.01921

[40] Xudong Wang, Xuanzhe Liu, Ying Zhang, and Gang Huang. 2012. Migration and Execution of JavaScript Applications Between Mobile Devices and Cloud. (2012), 83–84. https://doi.org/10.1145/2384716.2384750

[41] Zheng Wang and J. Crowcroft. 1996. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on Selected Areas in Communications* 14, 7 (Sep 1996), 1228–1234. https://doi.org/10.1109/49.536364

[42] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus. 2010. Deconvolutional networks. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2528–2535. https://doi.org/10.1109/CVPR.2010.5539957

[43] Ying Zhang, Gang Huang, Xuanzhe Liu, and others. 2012. Refactoring Android Java Code for On-demand Computation Offloading. *SIGPLAN Not.* 47, 10 (Oct. 2012), 233–248. https://doi.org/10.1145/2398857.2384634